augpy

Joachim Folz

Feb 20, 2021

CONTENTS

augpy is a lightweight library with minimal dependencies that provides a comprehensive tensor implementation for Cuda-enabled GPUs, with most of the functionality you are used to from numpy and Pytorch with a similar syntax.

What sets augpy apart is its focus on saturating math (no under or overflows possible), as well as comprehensive support for all data types in all functions. For example, augpy allows you to fill a `uint8` tensor with Gaussian distributed random numbers.

augpy's tensors are based on the *DLPack* specification and can be exchanged copy-free with other frameworks such as Jax or Pytorch.

While augpy's support for tensors is quite generic, it also includes additional functionality to work on 2D images, such as high quality affine warps with supersampling and Gaussian blurs.

CHAPTER

ONE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

TWO

EXAMPLES

2.1 Examples

Here are some simple examples of what you can do with augpy tensors.

Create a new tensor and do basic math on it:

```
>>> import augpy
>>> t = augpy.CudaTensor((3, 3)).fill(7)
>>> print(t.numpy())
[[7 7 7]
 [7 7 7]
 [7 7 7]]
>>> t += 11
>>> print(t.numpy())
[[18 18 18]
 [18 18 18]
 [18 18 18]]
```

Decode a JPEG image on the GPU:

```
>>> with open('cat.jpg', 'rb') as f:
...     data = f.read()
...
>>> decoder = augpy.Decoder()
>>> img = decoder.decode(data)
>>> print(img)
<CudaTensor shape=(480, 640, 3), device=0, dtype=uint8>
```

Convert a numpy array to augpy tensor:

```
>>> import numpy as np
>>> array = np.array([124, 116, 104], dtype=np.uint8)
>>> background = augpy.array_to_tensor(array)
>>> print(background)
<CudaTensor shape=(3), device=0, dtype=uint8>
```

Apply affine warp to an image:

```
>>> warped = augpy.CudaTensor((3, 224, 224))
>>> m, s = augpy.make_transform(img.shape[:2], warped.shape[1:], angle=10)
>>> print(m)
[[  2.1103022   -0.37210324 125.3217    ]
 [  0.37210324   2.1103022  -38.029423  ]]
```

(continues on next page)

(continued from previous page)

```
>>> augpy.warp_affine(img, warped, m, background, s)
>>> print(warped.numpy())
[[[124 124 124 ... 138 138 138]
  [124 124 124 ... 138 138 138]
  [124 124 124 ... 138 138 137]
  ...
  [110 106 102 ... 124 124 124]
  [101  95  90 ... 124 124 124]
  [ 86  82  77 ... 124 124 124]]

 [[116 116 116 ... 101 101 101]
  [116 116 116 ... 101 101 101]
  [116 116 116 ... 100 101 100]
  ...
  [ 94  91  90 ... 116 116 116]
  [ 88  86  83 ... 116 116 116]
  [ 82  78  75 ... 116 116 116]]

 [[104 104 104 ...  68  68  68]
  [104 104 104 ...  68  68  68]
  [104 104 104 ...  67  68  67]
  ...
  [ 85  84  83 ... 104 104 104]
  [ 83  81  80 ... 104 104 104]
  [ 81  80  77 ... 104 104 104]]]
```

Add noise to red channel:

```
>>> gen = augpy.RandomNumberGenerator()
>>> noise = augpy.CudaTensor((224, 224), dtype=augpy.int8)
>>> gen.gaussian(noise, 5, 80)
>>> print(noise.numpy())
[[ -15   20   -3 ...    4   40   22]
 [  50   73  -30 ...   -8   45   58]
 [ -65  -32  -32 ...  127   35  -76]
 ...
 [  79  -23   94 ...   43  -17  127]
 [   0  -24   -4 ... -128   27   25]
 [  -1  -21  127 ...   52 -128  -34]]
>>> augpy.fma(0.5, noise, warped[0], warped[0])
>>> print(warped.numpy())
[[[116 134 122 ... 140 158 149]
  [149 160 109 ... 134 160 167]
  [ 92 108 108 ... 202 156  99]
  ...
  [150  94 149 ... 146 116 188]
  [101  83  88 ...  60 138 136]
  [ 86  72 140 ... 150  60 107]]

 [[116 116 116 ... 101 101 101]
  [116 116 116 ... 101 101 101]
  [116 116 116 ... 100 101 100]
  ...
  [ 94  91  90 ... 116 116 116]
  [ 88  86  83 ... 116 116 116]
  [ 82  78  75 ... 116 116 116]]
```

(continues on next page)

(continued from previous page)

```
 [[104 104 104 ...  68  68  68]
  [104 104 104 ...  68  68  68]
  [104 104 104 ...  67  68  67]
  ...
  [ 85  84  83 ... 104 104 104]
  [ 83  81  80 ... 104 104 104]
  [ 81  80  77 ... 104 104 104]]]
```

Note: No clipping is required, since integer math is saturating. Over or underflows cannot occur.

Finally, export tensors to other frameworks like Pytorch:

```
>>> from torch.utils.dlpack import from_dlpack
>>> capsule = augpy.export_dltensor(warped)
>>> torch_tensor = from_dlpack(capsule)
>>> print(torch_tensor)
tensor([[[116, 134, 122,  ..., 140, 158, 149],
         [149, 160, 109,  ..., 134, 160, 167],
         [ 92, 108, 108,  ..., 202, 156,  99],
         ...,
         [150,  94, 149,  ..., 146, 116, 188],
         [101,  83,  88,  ...,  60, 138, 136],
         [ 86,  72, 140,  ..., 150,  60, 107]],

        [[116, 116, 116,  ..., 101, 101, 101],
         [116, 116, 116,  ..., 101, 101, 101],
         [116, 116, 116,  ..., 100, 101, 100],
         ...,
         [ 94,  91,  90,  ..., 116, 116, 116],
         [ 88,  86,  83,  ..., 116, 116, 116],
         [ 82,  78,  75,  ..., 116, 116, 116]],

        [[104, 104, 104,  ...,  68,  68,  68],
         [104, 104, 104,  ...,  68,  68,  68],
         [104, 104, 104,  ...,  67,  68,  67],
         ...,
         [ 85,  84,  83,  ..., 104, 104, 104],
         [ 83,  81,  80,  ..., 104, 104, 104],
         [ 81,  80,  77,  ..., 104, 104, 104]]], device='cuda:0',
       dtype=torch.uint8)
```

CHAPTER

THREE

PYTHON REFERENCE

3.1 Python Reference

3.1.1 Core Functionality

Exceptions raised by augpy, managing devices and computation streams, and controlling how functions are run on GPUs.

3.1.1.1 Device Management

augpy gives you fine control over which Cuda device is used and which Cuda stream kernels are run on. All functions are asynchronous by design, so events and streams can be used to synchronize host code.

There are two thread-local global variables that control which device and stream are currently active.

current_device

Each thread tracks its currently used Cuda device in the `current_device` variable. Use `CudaDevice.activate()` to make a stream the current stream and `CudaDevice.deactivate()`. to restore the previous state. Use `get_current_device()` to get the currently active device.

current_stream

Each thread tracks its currently used Cuda stream in the *current_stream* variable. Use *CudaStream.activate()* to make a stream the current stream and *CudaStream.deactivate()*. to restore the previous state. Use *get_current_stream()* to get the currently active stream.

default_stream

You can use the *default_stream* to synchronize CPU and GPU execution without explicitly creating and activating a different stream.

Note: All operations in augpy are asynchronous with respect to the CPU, so calling `CudaTensor.numpy()` will initiate copying data from the device to the host memory and return immediately. You need to use `CudaStream.synchronize()`, or `CudaEvent.record()` and `CudaEvent.synchronize()` to ensure that data is fully copied before the array is accessed.

---

class augpy.**CudaDevice**(*device_id: int*)

Create a new CudaDevice with the given Cuda device ID. 0 is the default and typically fastest device in the system.

Parameters **device_id** (*int*) – GPU device ID

__init__(*self: augpy._augpy.'CudaDevice'*, *device_id: int*) → None

Return type None

activate(*self: augpy._augpy.'CudaDevice'*) → None

Make this the *current_stream* and remember the previous stream.

Return type None

deactivate(*self: augpy._augpy.'CudaDevice'*) → None

Make the previous stream the *current_stream*.

Return type None

get_device(*self: augpy._augpy.'CudaDevice'*) → int

Return the device ID.

Return type int

get_properties(*self: augpy._augpy.'CudaDevice'*) → augpy._augpy.'CudaDevice'Prop

Return the device properties, see py/core:get_device_properties for more detials.

Return type 'CudaDevice'Prop

synchronize(*self: augpy._augpy.'CudaDevice'*) → None

Block until all work on this device has finished. Cuda uses busy waiting to achieve this. See synchronization method of py/core:CudaStream or py/core:CudaEvent to avoid the CPU load this incurs.

Return type None

class augpy.**CudaEvent**

Convenience wrapper for the cudaEvent_t.

Creating a new CudaEvent retrieves an event from the event pool of the *current_device*.

__init__(*self: augpy._augpy.'CudaEvent'*) → None

Return type None

query (*self: augpy._augpy.'CudaEvent'*) → bool
Returns `True` if event has occurred.

Return type bool

record (*self: augpy._augpy.'CudaEvent'*) → None
Record wrapped event on *current_stream*.

Return type None

synchronize (*self: augpy._augpy.'CudaEvent'*, *microseconds: int = 100*) → None
Block until event has occurred. Checks in `microseconds` interval. Faster intervals make this more accurate, but increase CPU load. Uses standard Cuda busy-waiting method if `microseconds <= 0`.

Parameters **microseconds** (*int*) – check interval

Return type None

class augpy.**CudaStream**(*device_id: int = 0*, *priority: int = 0*)
Convenience wrapper for the cudaStream_t type.

Creates a new Cuda stream on the given device. Lower numbers mean higher priority, and values are clipped to the valid range. Use *get_device_properties()* to get the range of possible values for a device.

See: cudaStreamCreateWithPriority

Use `device_id=-1` and `priority=-1` to get the *default_stream*.

Parameters

- **device_id** (*int*) – GPU device ID
- **priority** (*int*) – stream priority

__init__ (*self: augpy._augpy.'CudaStream'*, *device_id: int = 0*, *priority: int = 0*) → None

Return type None

activate (*self: augpy._augpy.'CudaStream'*) → None
Make this the *current_stream* and remember the previous stream.

Return type None

deactivate (*self: augpy._augpy.'CudaStream'*) → None
Make the previous stream the *current_stream*.

Return type None

synchronize (*self: augpy._augpy.'CudaStream'*, *microseconds: int = 100*) → None
Block until all work on this stream has finished. Checks in `microseconds` interval. Faster intervals make this more accurate, but increase CPU load. Uses standard Cuda busy-waiting method if `microseconds <= 0`.

Return type None

augpy.**get_current_device**() → int
Returns the active device ID.

See: *current_device*.

rtype int

augpy.**get_current_stream**() → augpy._augpy.CudaStream
Returns the active *CudaStream*.

See: *current_stream*

rtype CudaStream

augpy.**default_stream**
The default *CudaStream*. Implicitly available on all Cuda devices.

augpy.**release**() → None
Release all allocated memory on all GPUs. All *CudaTensors* become invalid immediately. Do I have to tell you this is dangerous?

Return type None

3.1.1.2 Device Information

augpy.**get_device_properties**(*device_id: int*) → augpy._augpy.CudaDeviceProp
Get *CudaDeviceProp* for given device.

Parameters **device_id** (*int*) – Cude device id

Returns properties of device

Return type *CudaDeviceProp*

class augpy.**CudaDeviceProp**
The cudaDeviceProp struct extended with stream priority fields *leastStreamPriority* and *greatestStreamPriority*, *coresPerMultiprocessor*, and *maxGridSize*.

__init__()
Initialize self. See help(type(self)) for accurate signature.

property coresPerMultiprocessor
Number of Cuda cores per multiprocessor

property coresPerSM
Number of Cuda cores per SM.

property greatestStreamPriority
Highest priority a Cuda stream on this device can have.

property l2CacheSize
Size of L2 cache in bytes

property leastStreamPriority
Lowest priority a Cuda stream on this device can have.

property major
Major compute capability

property maxGridSize
Max number of blocks in each grid dimension

property maxThreadsDim
Maximum size of each dimension of a block

property maxThreadsPerBlock
Maximum number of threads per block

property maxThreadsPerMultiProcessor
Maximum resident threads per multiprocessor

property minor
Minor compute capability

property multiProcessorCount
Number of multiprocessors on device

property name
ASCII string identifying device

property numCudaCores
Total number of Cuda coes.

property regsPerBlock
32-bit registers available per block

property regsPerMultiprocessor
32-bit registers available per multiprocessor

property sharedMemPerBlock
Shared memory available per block in bytes

property sharedMemPerMultiprocessor
Shared memory available per multiprocessor in bytes

property streamPrioritiesSupported
Device supports stream priorities

property totalConstMem
Constant memory available on device in bytes

property totalGlobalMem
Global memory available on device in bytes

property warpSize
Warp size in threads

augpy.**meminfo**(*device_id: int = 0*) → Tuple[int, int, int]
For the device defined by `device_id`, return the current used, free, and total memory in bytes.

Return type Tuple[int, int, int]

3.1.1.3 Exceptions

exception augpy.**CudaError**
Raised when a problem with a GPU occurs, e.g., device is unavailable or invalid kernel configuration.

exception augpy.**CuRandError**
Raised when a problem with CuRand occurs, e.g., no memory left for random state.

exception augpy.**CuBlasError**
Raised when a problem with CuBlas occurs, e.g., no memory left for handle.

exception augpy.**MemoryError**
Raised when a problem with GPU memory occurs, e.g., no memory left for tensor.

exception augpy.**NvJpegError**
Raised when a problem with JPEG decoding occurs, e.g., corrupt or unsupported image.

3.1.1.4 Blocks and threads

Cuda code executes in blocks of threads, each of which calculates one or more values in a tensor. The number of threads in a block greatly influences the performance of kernels, as they will share resources likes caches, but can also collaborate in calculations.

A Cuda-enabled GPU is organized in SMs with a number of Cuda cores each. Each SM can work on a block independently. Thus, it is important that a task is divided into at least as many blocks as there are SMs. You can use *`get_device_properties()`* to find out, among other useful information, how many SMs there are and the number of Cuda cores per SM.

augpy functions often allow you to control how they are executed on the GPU:

1. Set the `blocks_per_sm` parameter to control how many blocks the work is divided into. The total number of blocks will be `blocks_per_sm` times number of SMs on the GPU.
2. Set the `threads` parameter to control how many threads there are in each block. If `threads` is zero, the number of cores per SM is used.

Together these parameters define the total number of threads that will be started for the kernel. The calculations for each element of the tensor are distributed evenly among these threads, i.e., each thread may calculate more than one value. More values per thread is often more efficient, however it must be balanced against the number of blocks to keep all SMs busy.

The defaults for `blocks_per_sm` and `threads` parameters are usually quite sensible, but depending on your GPU architecture other combinations may provide better performance.

3.1.2 Tensors

Note: All operations in augpy are asynchronous with respect to the CPU, i.e., function call initiate work on the GPU and return immediately. For example `CudaTensor.numpy()` will initiate copying data from the device to the host memory and return the array immediately, even though data has not yet been fully copied over.

Use `CudaStream.synchronize()`, or `CudaEvent.record()` and `CudaEvent.synchronize()` to synchronize CPU code with the respective stream or event on the GPU.

However, all work done on the GPU is sequential within a `CudaStream`. You can use augpy functions to "queue up" operations on tensors, so synchronization is only required when using interacting with the CPU or another GPU framework.

3.1.2.1 Data Types

class augpy.**DLDataType**(*code: int*, *bits: int*, *lanes: int = 1*)
Bases: augpy._augpy.pybind11_object

DLPack data type for *CudaTensors*.

Parameters

- **code** (*int*) – See *DLDataTypeCode*
- **bits** (*int*) – Number of bits
- **lanes** (*int*) – Number of elements for vector types; must be 1 to use with *CudaTensor*

__init__(*self: augpy._augpy.'DLDataType'*, *code: int*, *bits: int*, *lanes: int = 1*) → None

Return type None

property bits
Number of bits.

property code
See 'DLDataType'Code.

property itemsize
Number of bytes per element with this data type.

property lanes
Mumber of elements for vector types. Must be 1 to use with *CudaTensor*.

class augpy.**DLDataTypeCode**(*arg0: int*)
Bases: object

DLPack type code enum.

Members:

kDLInt : Signed integer.

kDLUInt : Unsigned integer.

kDLFloat : Floating point number.

__init__(*self: augpy._augpy.'DLDataTypeCode'*, *arg0: int*) → None

Return type None

property kDLFloat
DLPack type code enum.

Members:

kDLInt : Signed integer.

kDLUInt : Unsigned integer.

kDLFloat : Floating point number.

property kDLInt
DLPack type code enum.

Members:

kDLInt : Signed integer.

kDLUInt : Unsigned integer.

kDLFloat : Floating point number.

property kDLUInt

DLPack type code enum.

Members:

kDLInt : Signed integer.

kDLUInt : Unsigned integer.

kDLFloat : Floating point number.

augpy.**to_augpy_dtype**(*numpy_dtype: Union[type, numpy.dtype]*) → augpy._augpy.DLDataType

Translate numpy to augpy data types.

Example:

```
to_augpy_dtype(numpy.uint8) == augpy.uint8
```

Parameters **numpy_dtype** (*Union[type, numpy.dtype]*) – numpy type to translate

Return type *DLDataType*

augpy.**to_numpy_dtype**(*augpy_dtype: augpy._augpy.DLDataType*) → type

Translate augpy to numpy data types.

Example:

```
to_numpy_dtype(augpy.uint8) == numpy.uint8
```

Parameters **augpy_dtype** (DLDataType) – augpy type to translate

Return type type

augpy.**swap_dtype**(*dtype: Union[augpy._augpy.DLDataType, type, numpy.dtype]*) → Union[augpy._augpy.DLDataType, type]

Translate to and from numpy and augpy data types.

Examples:

```
swap_dtype(augpy.uint8) == numpy.uint8
swap_dtype(numpy.uint8) == augpy.uint8
```

Parameters **dtype** (*Union[DLDataType, type, numpy.dtype]*) – type to translate to augpy or numpy

Return type Union[*DLDataType*, type]

augpy.**to_temp_dtype**(*dtype: Union[augpy._augpy.DLDataType, type, numpy.dtype]*) → Union[augpy._augpy.DLDataType, type]

augpy defines a temp type for each tensor data type. This temp type is used internally for processing and sometimes returns. This dict maps from augpy and numpy dtypes and their temp types.

This function returns the temp type for given data type.

Parameters **dtype** (*Union[DLDataType, type, numpy.dtype]*) – augpy or numpy dtype

Returns temp type

Return type Union[*DLDataType*, type]

augpy.**int8**
<DLDataType int8>

augpy.**int16**
<DLDataType int16>

augpy.**int32**
<DLDataType int32>

augpy.**int64**
<DLDataType int64>

augpy.**uint8**
<DLDataType uint8>

augpy.**uint16**
<DLDataType uint16>

augpy.**uint32**
<DLDataType uint32>

augpy.**uint64**
<DLDataType uint64>

augpy.**float16**
<DLDataType float16>

> **Warning:** Not yet supported.

augpy.**float32**
<DLDataType float32>

augpy.**float64**
<DLDataType float64>

3.1.2.2 CudaTensor

augpy's tensor class. It is a backwards compatible extension to the *DLPack* specification.

It supports all the usual operations you would expect from a full-featured tensor class, like complex indexing and slicing:

```
>>> t = CudaTensor((2, 2, 4), uint8)
>>> t
<CudaTensor shape=(2, 2, 4), device=0, dtype=uint8>
>>> t[1, 1, 3]
<CudaTensor shape=(), device=0, dtype=uint8>
>>> t[-1]
<CudaTensor shape=(2, 4), device=0, dtype=uint8>
>>> t[:,0]
<CudaTensor shape=(2, 4), device=0, dtype=uint8>
>>> t[:, 1:, 1:-1]
<CudaTensor shape=(2, 1, 2), device=0, dtype=uint8>
>>> t[:, :, ::2]
<CudaTensor shape=(2, 2, 2), device=0, dtype=uint8>
```

Math and comparison operations you are used to from numpy or Pytorch also work just fine:

```
>>> t.numpy()
array([[[0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 0, 0, 0]]], dtype=uint8)
>>> t += 3
>>> t.numpy()
array([[[3, 3, 3, 3],
        [3, 3, 3, 3]],

       [[3, 3, 3, 3],
        [3, 3, 3, 3]]], dtype=uint8)
>>> (5 - t).numpy()
array([[[2, 2, 2, 2],
        [2, 2, 2, 2]],

       [[2, 2, 2, 2],
        [2, 2, 2, 2]]], dtype=uint8)
>>> (t > (t - 1)).numpy()
array([[[1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=uint8)
```

Note: All operations in augpy are asynchronous, so calling `CudaTensor.numpy()` will initiate copying data from the device to the host memory. You need to use `CudaStream.synchronize()`, or `CudaEvent.record()` and `CudaEvent.synchronize()` to ensure that data is fully copied before the array is accessed.

Math is saturating in augpy. Integer tensors will never over or underflow:

```
>>> t[:] = 40
>>> t.numpy()
array([[[40, 40, 40, 40],
        [40, 40, 40, 40]],

       [[40, 40, 40, 40],
        [40, 40, 40, 40]]], dtype=uint8)
>>> (0 - t).numpy()
array([[[0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 0, 0, 0]]], dtype=uint8)
```

Broadcasting is also supported:

```
>>> t1 = CudaTensor((3, 1), uint8)
>>> t2 = CudaTensor((1, 3), uint8)
>>> ((t1 + 3) * (t2 + 4)).numpy()
array([[12, 12, 12],
       [12, 12, 12],
       [12, 12, 12]], dtype=uint8)
```

Note: Tensors may appear to be initialized with zeros. They may, however, reuse memory from previously deleted tensors, so they should be treated as uninitialized and need to be zeroed or otherwise initialized.

class augpy.**CudaTensor**(*shape: List[int], dtype: augpy._augpy.DLDataType = <augpy._augpy.DLDataType object>, device_id: int = 0*)

Bases: augpy._augpy.pybind11_object

Create a new, empty tensor on a GPU device.

Parameters

- **shape** (*List[int]*) – shape of the tensor
- **dtype** (DLDataType) – data type
- **device_id** (*int*) – Cuda device id

__init__(*self: augpy._augpy.'CudaTensor', shape: List[int], dtype: augpy._augpy.DLDataType = DLDataType(code=kDLUInt, bits=8), device_id: int = 0*) → None

Return type None

property byte_offset

Starting offset in bytes for the data pointer.

property dtype

Tensor data type.

fill(**args*, ***kwargs*)

fill(*self: 'CudaTensor', scalar: float*) → 'CudaTensor'

Fill the tensor with the given scalar value.

Returns this tensor

Return type """'CudaTensor"""

fill(*self: 'CudaTensor', other: 'CudaTensor'*) → 'CudaTensor'

Copy the given tensor into this tensor.

Returns this tensor

Return type """'CudaTensor"""

property is_contiguous

`True` if the tensor is contiguous, i.e., elements are located next to each other in memory.

property itemsize

Size of the one element in bytes.

property ndim

Number of dimensions.

numpy(**args*, ***kwargs*)

numpy(*self: 'CudaTensor'*) → array

Create a new numpy array and start copying data from the device to host memory.

Return type array

numpy(*self: 'CudaTensor', array: buffer = None*) → array

Create a new numpy array from the given buffer and start copying data from the device to host memory.

Parameters **array** (*buffer*) – buffer to create new array from

Return type array

property ptr
Data pointer.

reshape (*self: augpy._augpy.'CudaTensor', shape: List[int]*) → augpy._augpy.'CudaTensor'
Return a new tensor that uses the same backing memory with a different shape. Shape must have same number of elements. Only contiguous tensors can be reshaped.

Parameters **shape** (*List[int]*) – new shape

Return type 'CudaTensor'

property shape
Tensor shape.

property size
Number of elements in the tensor.

property strides
Tensor strides, i.e., the number of elements to add to a flat tensor to reach the next element for each dimension.

sum (**args*, ***kwargs*)

sum (*self: 'CudaTensor'*, *upcast: bool = False*) → 'CudaTensor'
Sum all values in the tensor.
Parameters **upcast** (*bool*) – if `True`, the output scalar tensor will be promoted to a more expressive data type to avoid saturation
Returns sum as scalar tensor
Return type """CudaTensor"""

sum (*self: 'CudaTensor'*, *axis: int*, *keepdim: bool = False*, *upcast: bool = False*, *out: 'CudaTensor' = None*, *blocks_per_sm: int = 8*, *threads: int = 0*) → 'CudaTensor'
Sum all values in the tensor along an axis.
Parameters
- **axis** (*int*) – which axis to sum along
- **keepdim** (*bool*) – keep the summed dimension with size 1
- **upcast** (*bool*) – if `True`, the output scalar tensor will be promoted to a more expressive data type to avoid saturation
- **out** (*"""CudaTensor"""*) – use this tensor as output, must have correct shape, and same data type if `upcast` is `False`, otherwise promoted type is required

Returns tensor summed along axis
Return type """CudaTensor"""

3.1.2.3 Creation & Conversion

augpy.**cast** (**args*, ***kwargs*)

augpy.**cast** (*tensor: CudaTensor*, *out: CudaTensor*, *blocks_per_sm: int = 8*, *threads: int = 0*) → CudaTensor
Read values from `tensor`, cast them to the data type of `out` and store them there. `tensor` and `out` must have the same shape.

Parameters
- **tensor** (CudaTensor) – source tensor
- **out** (CudaTensor) – output tensor

Return type *CudaTensor*

augpy.**cast**(*tensor: CudaTensor*, *dtype: DLDataType*, *blocks_per_sm: int = 8*, *threads: int = 0*) → CudaTensor

Create a new tensor with values from `tensor` cast to the given data type `dtype`.

Parameters

- **tensor** (`CudaTensor`) – source tensor
- **dtype** (`DLDataType`) – target data type

Returns new tensor with given data type

Return type *CudaTensor*

augpy.**copy**(*src: augpy._augpy.CudaTensor*, *dst: augpy._augpy.CudaTensor*, *blocks_per_sm: int = 8*, *threads: int = 0*) → augpy._augpy.CudaTensor

Copy `src` into `dst`. Supports broadcasting.

Return type *CudaTensor*

augpy.**empty_like**(*tensor: augpy._augpy.CudaTensor*) → augpy._augpy.CudaTensor

Create a new tensor with the same shape, dtype and on the same device as `tensor`.

Return type *CudaTensor*

augpy.**array_to_tensor**(**args*, ***kwargs*)

augpy.**array_to_tensor**(*array: buffer*, *device_id: int = 0*) → CudaTensor

Copy a Python buffer into a new tensor on the specified GPU device. This initiates an asynchronous copy from host to device memory.

Return type *CudaTensor*

augpy.**array_to_tensor**(*array: buffer*, *tensor: CudaTensor*) → CudaTensor

Copy a Python buffer to a tensor created from the given buffer `tensor`. This initiates an asynchronous copy from host to device memory.

Return type *CudaTensor*

augpy.**tensor_to_array**(**args*, ***kwargs*)

augpy.**tensor_to_array**(*tensor: CudaTensor*) → array

Copy a given tensor to a new numpy array. This initiates an asynchronous copy from device to host memory.

Return type array

augpy.**tensor_to_array**(*tensor: CudaTensor*, *array: buffer*) → array

Copy a given tensor to a numpy array created from the given buffer `array`. This initiates an asynchronous copy from device to host memory.

Return type array

augpy.**import_dltensor**(*tensor_capsule: capsule*, *name: str*) → augpy._augpy.CudaTensor

Import a GPU tensor from another library into augpy.

Parameters

- **tensor_capsule** (*capsule*) – a Python capsule object that contains a *DLManagedTensor*
- **name** (*str*) – name under which the tensor is stored in the capsule, e.g., `"dltensor"` for Pytorch

Returns other tensor wrapped in a *CudaTensor*

Return type *CudaTensor*

augpy.**export_dltensor**(*tensor: object*, *name: str = 'dltensor'*, *destruct: bool = True*) → capsule
Export a GPU tensor to be used by another library.

Parameters

- **pytensor** – Python-wrapped CudaTensor
- **name** (*str*) – name under which the tensor is stored in the returned capsule, e.g., *"dltensor"* for Pytorch
- **destruct** (*bool*) – if `True`, add a destructor to the capsule which will delete the tensor when the capsule is deleted; only set to `False` if you know what you're doing

Returns capsule with exported *CudaTensor*

Return type capsule

3.1.2.4 Functions

augpy.**add**(**args*, ***kwargs*)

augpy.**add**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor
Add a `scalar` value to a `tensor`.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**add**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor
Add `tensor2` to `tensor1`.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**sub**(**args*, ***kwargs*)

augpy.**sub**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor
Subtract a `scalar` value from a `tensor`.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**sub**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Subtract `tensor2` from `tensor1`.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**rsub**(*tensor: augpy._augpy.CudaTensor*, *scalar: float*, *out: augpy._augpy.CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → augpy._augpy.CudaTensor

Subtract a `tensor` from a `scalar` value.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**mul**(**args*, ***kwargs*)

augpy.**mul**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Multiply a `tensor` by a `scalar` value.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**mul**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Multiply `tensor1` by `tensor2`.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor

- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**div**(**args*, ***kwargs*)

augpy.**div**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Divide a `tensor` by a `scalar` value.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**div**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Divide *tensor1* by *tensor2*.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**rdiv**(*tensor: augpy._augpy.CudaTensor*, *scalar: float*, *out: augpy._augpy.CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → augpy._augpy.CudaTensor

Divide a `scalar` value by a `tensor`.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**lt**(**args*, ***kwargs*)

augpy.**lt**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Compute `tensor < scalar` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value

- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**lt**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Compute `tensor1 >= tensor2` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**le**(**args*, ***kwargs*)

augpy.**le**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Compute `tensor <= scalar` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**le**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Compute `tensor1 >= tensor2` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**gt**(*tensor: augpy._augpy.CudaTensor*, *scalar: float*, *out: augpy._augpy.CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → augpy._augpy.CudaTensor

Compute `tensor > scalar` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value

- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**ge**(*tensor: augpy._augpy.CudaTensor*, *scalar: float*, *out: augpy._augpy.CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → augpy._augpy.CudaTensor

Compute `tensor >= scalar` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**eq**(**args*, ***kwargs*)

augpy.**eq**(*tensor: CudaTensor*, *scalar: float*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Compute `tensor == scalar` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor** (CudaTensor) – tensor
- **scalar** (*float*) – scalar value
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**eq**(*tensor1: CudaTensor*, *tensor2: CudaTensor*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → CudaTensor

Compute `tensor1 == tensor2` as `uint8` tensor, where `1` means the condition is met and `0` otherwise.

Parameters

- **tensor1** (CudaTensor) – first tensor
- **tensor2** (CudaTensor) – second tensor
- **out** (CudaTensor) – optional output tensor

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**fma**(*scalar: float*, *tensor1: augpy._augpy.CudaTensor*, *tensor2: augpy._augpy.CudaTensor*, *out: augpy._augpy.CudaTensor = None*, *blocks_per_sm: int = 8*, *threads: int = 512*) → augpy._augpy.CudaTensor

Compute a fused multiply-add on a scalar and two tensors, i.e.,

$$r = s \cdot t_1 \cdot t_2$$

If `tensor1` has an unsigned integer data type, then `tensor2` must have the signed version of the same type, e.g., a `uint8` tensor must be paired with a `int8` tensor.

Parameters

- **scalar** (*float*) – scalar factor
- **tensor1** (CudaTensor) – tensor t_1
- **tensor2** (CudaTensor) – tensor t_2
- **out** (CudaTensor) – optional output tensor r

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**gemm**(*A: augpy._augpy.CudaTensor*, *B: augpy._augpy.CudaTensor*, *C: augpy._augpy.CudaTensor = None*, *alpha: float = 1.0*, *beta: float = 0.0*) → augpy._augpy.CudaTensor

Calculate the matrix multiplication of two 2D tensors. More specifically calculates

$$C = Aimes(lpha \cdot B) + eta \cdot C$$

Only `float` and `double` are supported.

All tensors must have the same data type.

All tensors must be contiguous.

Returns new output tensor if `C` is `None`, otherwise `C`

Return type *CudaTensor*

augpy.**fill**(*scalar: float*, *dst: augpy._augpy.CudaTensor*, *blocks_per_sm: int = 8*, *threads: int = 0*) → augpy._augpy.CudaTensor

Fill *src* with the given *scalar* value.

Return type *CudaTensor*

augpy.**sum**(**args*, ***kwargs*)

augpy.**sum**(*tensor: CudaTensor*, *upcast: bool = False*) → CudaTensor

Sum all elements in a tensor with saturation.

Parameters

- **tensor** (CudaTensor) – tensor to sum, must be contiguous
- **upcast** (*bool*) – if `True`, returns tensor with `float` or `double` type

Returns sum value as scalar tensor

Return type *CudaTensor*

augpy.**sum**(*tensor: CudaTensor*, *axis: int*, *keepdim: bool = False*, *upcast: bool = False*, *out: CudaTensor = None*, *blocks_per_sm: int = 8*, *num_threads: int = 0*) → CudaTensor

Sum of all elements along an axis in a tensor with saturation.

Parameters

- **tensor** (CudaTensor) – tensor to sum, may be strided
- **axis** (*int*) – axis index to sum along
- **keepdim** (*bool*) – if `True`, keep sum axis dimension with length 1
- **upcast** (*bool*) – if `True`, returns tensor with `float` or `double` type
- **out** (CudaTensor) – output tensor (may be `None`)

Returns tensor with values summed along axis, or `None` if `out` is tensor

Return type *CudaTensor*

3.1.3 Random Number Generation

Classes for random number generation. augpy's speciality is that all dtypes are supported for all distributions. For example, it is possible to fill an integer tensor with approximately Gaussian distributed numbers.

class augpy.**RandomNumberGenerator**(*device_id: object = None*, *seed: object = None*)
Bases: augpy._augpy.pybind11_object

A convenient wrapper for cuRAND methods that fill tensors with pseudo-random numbers.

Parameters

- **device_id** (*object*) – GPU device ID; if None, *current_device* is used
- **seed** (*object*) – random seed; if None, read values from std::random_device to create a random seed.

__init__(*self: augpy._augpy.'RandomNumberGenerator'*, *device_id: object = None*, *seed: object = None*) → None

Return type None

gaussian(*self: augpy._augpy.'RandomNumberGenerator'*, *target: augpy._augpy.CudaTensor*, *mean: float = 0.0*, *std: float = 1.0*, *blocks_per_sm: int = 8*, *threads: int = 0*) → None
Fill target tensor with Gaussian distributed numbers with specified mean and standard deviation std.

Note: This is supported for integer tensors. Values are drawn from the given distribution, then rounded and cast to the data type of the tensor with saturation. The values in an integer tensor are thus only approximately Gaussian distributed.

Parameters

- **target** (CudaTensor) – tensor to fill
- **mean** (*float*) – Gaussian mean
- **std** (*float*) – Gaussian standard deviation

Return type None

uniform(*self: augpy._augpy.'RandomNumberGenerator'*, *target: augpy._augpy.CudaTensor*, *vmin: float*, *vmax: float*, *blocks_per_sm: int = 8*, *threads: int = 0*) → None

Fill target tensor with uniformly distributed number in $[v_{min}, v_{max})$.

This is supported for integer tensors. Values are cast from float or double down to the integer type. The mean of the values is approximately :math:`

rac{v_{max} + v_{min}}{2}`.

Saturation is not used. v_{min} and v_{max} must be representable in the target tensor data type, else values may under or overflow.

Parameters:

target: tensor to fill vmin: minimum value; can occur vmax: maximum value; does not occur

rtype None

3.1.4 Image Functions

3.1.4.1 JPEG Decoding

Hybrid JPEG decoding on CPU and GPU using Nvjpeg.

class augpy.**Decoder**(*device_padding: int = 16777216*, *host_padding: int = 8388608*, *gpu_huffman: bool = False*)

Bases: augpy._augpy.pybind11_object

Wrapper for Nvjpeg-based JPEG decoding, created on the *current_device*.

See: Nvjpeg docs

Parameters

- **device_padding** (*int*) – memory padding on the device
- **host_padding** (*int*) – memory padding on the host
- **gpu_huffman** (*bool*) – enable Huffman decoding on the GPU; not recommended unless you really need to offload from CPU

__init__(*self: augpy._augpy.'Decoder'*, *device_padding: int = 16777216*, *host_padding: int = 8388608*, *gpu_huffman: bool = False*) → None

Return type None

decode(*self: augpy._augpy.'Decoder'*, *data: str*, *buffer: augpy._augpy.CudaTensor = None*) → augpy._augpy.CudaTensor

Decode a JPEG image using Nvjpeg. Output is in (H, W, C) format and resides on the GPU device.

Parameters

- **data** (*str*) – compressed JPEG image as a JFIF string, i.e., the full file contents
- **buffer** (CudaTensor) – optional buffer to use; may be None; if not None must be big enough to contain the decoded image

Returns new tensor with decoded image on GPU in (H, W, C) format

Return type *CudaTensor*

3.1.4.2 Affine Warp

Functions to apply affine transformations on 2D images.

augpy.**make_transform**(*source_size: Tuple[int, int]*, *target_size: Tuple[int, int]*, *angle: float = 0*, *scale: float = 1*, *aspect: float = 1*, *shift: Optional[Tuple[float, float]] = None*, *shear: Optional[Tuple[float, float]] = None*, *hmirror: bool = False*, *vmirror: bool = False*, *scale_mode: Union[str, augpy._augpy.WarpScaleMode] = <augpy._augpy.WarpScaleMode object>*, *max_supersampling: int = 3*, *out: Optional[numpy.ndarray] = None*, *__template__=array([[0., 0., 0.], [0., 0., 0.]], dtype=float32)*, ***__*) → Tuple[numpy.ndarray, int]

Convenience wrapper for *make_affine_matrix()*.

See: *make_affine_matrix()* slightly faster, less convenient version of this function.

Parameters

- **source_size** (*Tuple[int, int]*) – source height (h_s) and width (w_s)

- **target_size** (*Tuple[int, int]*) – target height (h_t) and width (w_t)
- **angle** (*float*) – clockwise angle in degrees with image center as rotation axis
- **scale** (*float*) – scale factor relative to output size; 1 means fill target height or width wise depending on `scale_mode` and whichever is longest/shortest; larger values will crop, smaller values leave empty space in the output canvas
- **aspect** (*float*) – controls the aspect ratio; 1 means same as input, values greater 1 increase the width and reduce the height
- **shift** (*Optional[Tuple[float, float]]*) – `(shifty, shiftx)` or `None` for `(0, 0)`; shift the image in y (vertical) and x (horizontal) direction; 0 centers the image on the output canvas; -1 means shift up/left as much as possible; 1 means shfit down/right as much as possible; the maximum distance to shift is $max(scale \cdot h_s - h_t, h_t - scale \cdot h_s)$
- **shear** (*Optional[Tuple[float, float]]*) – `(sheary, shearx)` or `None` for `(0, 0)`; controls up/down and left/right shear; for every pixel in the x direction move `sheary` pixels in y direction, same for y direction
- **hmirror** (*bool*) – if `True` flip image horizontally
- **vmirror** (*bool*) – if `True` flip image vertically
- **scale_mode** (*Union[str, WarpScaleMode]*) – if *WarpScaleMode.WARP_SCALE_SHORTEST* scale is relative to shortest side; this fills the output canvas, cropping the image if necessary; if *WarpScaleMode.WARP_SCALE_LONGEST* scale is relative to longest side; this ensures the image is contained inside the output canvas, but leaves empty space
- **max_supersampling** (*int*) – upper limit for recommended supersampling
- **out** (*Optional[numpy.ndarray]*) – optional 2×3 `float` output array

Returns transformation matrix and suggested supersampling factor

Return type Tuple[numpy.ndarray, int]

augpy.**make_affine_matrix**(*out: buffer*, *source_height: int*, *source_width: int*, *target_height: int*, *target_width: int*, *angle: float = 0.0*, *scale: float = 1.0*, *aspect: float = 1.0*, *shifty: float = 0.0*, *shiftx: float = 0.0*, *sheary: float = 0.0*, *shearx: float = 0.0*, *hmirror: bool = False*, *vmirror: bool = False*, *scale_mode: augpy._augpy.WarpScaleMode = WarpScaleMode.WARP_SCALE_SHORTEST*, *max_supersampling: int = 3*) → int

Create a $2imes3$ matrix for a set of affine transformations. This matrix is compatible with the warpAffine function of OpenCV with the WARP_INVERSE_MAP flag set.

Transforms are applied in the following order:

1. shear
2. scale & aspect ratio
3. horizontal & vertical mirror
4. rotation
5. horizontal & vertical shift

See: *make_transform()* for a more convenient version of this function.

Parameters

- **out** (*buffer*) – output buffer that matrix is written to; must be a writeable $2imes3$ `float` buffer
- **source_height** (*int*) – h_s height of the image in pixels
- **source_width** (*int*) – w_s width of the image in pixels
- **target_height** (*int*) – h_t height of the output canvas in pixels
- **target_width** (*int*) – w_t width of the output canvas in pixels
- **angle** (*float*) – clockwise angle in degrees with image center as rotation axis
- **scale** (*float*) – scale factor relative to output size; 1 means fill target height or width wise depending on `scale_mode` and whichever is longest/shortest; larger values will crop, smaller values leave empty space in the output canvas
- **aspect** (*float*) – controls the aspect ratio; 1 means same as input, values greater 1 increase the width and reduce the height
- **shifty** (*float*) – shift the image in y direction (vertical); 0 centers the image on the output canvas; -1 means shift up as much as possible; 1 means shfit down as much as possible; the maximum distance to shift is $max(scale \cdot h_s - h_t, h_t - scale \cdot h_s)$
- **shiftx** (*float*) – same as `shifty`, but in x direction (horizontal)
- **sheary** (*float*) – controls up/down shear; for every pixel in the x direction move `sheary` pixels in y direction
- **shearx** (*float*) – same as `sheary` but controls left/right shear
- **hmirror** (*bool*) – if `True` flip image horizontally
- **vmirror** (*bool*) – if `True` flip image vertically
- **scale_mode** (`WarpScaleMode`) – if *WarpScaleMode.WARP_SCALE_SHORTEST* scale is relative to shortest side; this fills the output canvas, cropping the image if necessary; if *WarpScaleMode.WARP_SCALE_LONGEST* scale is relative to longest side; this ensures the image is contained inside the output canvas, but leaves empty space
- **max_supersampling** (*int*) – upper limit for recommended supersampling

Returns recommended supersampling factor for the warp

Return type int

augpy.**warp_affine**(*src: augpy._augpy.CudaTensor*, *dst: augpy._augpy.CudaTensor*, *matrix: buffer*, *background: augpy._augpy.CudaTensor*, *supersampling: int*) → None

Takes an image in channels-last format (H, W, C) and affine warps it into a given output tensor in channels-first format (C, H, W). Any blank canvas is filled with a background color. The warp is performed with bi-linear and supersampling.

Parameters

- **src** (`CudaTensor`) – image tensor
- **dst** (`CudaTensor`) – target tensor
- **matrix** (*buffer*) – $2imes3$ `float` transformation matrix, see *make_affine_matrix()* for details
- **background** (`CudaTensor`) – background color to fill empty canvas
- **supersampling** (*int*) – supersampling factor, e.g., 3 means 9 samples are taken in a $3imes3$ grid

Return type None

class augpy.**WarpScaleMode**(*arg0: int*)

Bases: object

Enum whether to scale relative to the shortest or longest side of the image.

Members:

WARP_SCALE_SHORTEST : Scaling is relative to the shortest side of the image.

WARP_SCALE_LONGEST : Scaling is relative to the longest side of the image.

property WARP_SCALE_LONGEST

Enum whether to scale relative to the shortest or longest side of the image.

Members:

WARP_SCALE_SHORTEST : Scaling is relative to the shortest side of the image.

WARP_SCALE_LONGEST : Scaling is relative to the longest side of the image.

property WARP_SCALE_SHORTEST

Enum whether to scale relative to the shortest or longest side of the image.

Members:

WARP_SCALE_SHORTEST : Scaling is relative to the shortest side of the image.

WARP_SCALE_LONGEST : Scaling is relative to the longest side of the image.

__init__(*self: augpy._augpy.'WarpScaleMode'*, *arg0: int*) → None

Return type None

3.1.4.3 Lighting

The following functions change the lighting of 2D images. Both input and output must be contiguous and in channel-first format (C, H, W) (channel, height, width). All dtypes and an arbitrary number of channels is supported.

Output tensor `out` may be `None`, in which case a new tensor of the same shape and dtype as the input is returned. Output tensor must be same shape and dtype as the input. If output is given `None` is returned.

augpy.**lighting**(*imtensor: augpy._augpy.CudaTensor*, *gammagrays: augpy._augpy.CudaTensor*, *gammacolors: augpy._augpy.CudaTensor*, *contrasts: augpy._augpy.CudaTensor*, *vmin: float*, *vmax: float*, *out: augpy._augpy.CudaTensor = None*) → augpy._augpy.CudaTensor

Apply lighting augmentation to a batch of images. This is a four-step process:

1. Normalize values :math:`v_{norm} =

rac{v - v_{min}}{v_{max}-v_{min}}`

with v_{max} the minimum and v_{max} the maximum lightness value

1. Apply contrast change
2. Apply gamma correction
3. Denormalize values $v' = v_{norm} * (v_{max} - v_{min}) + v_{min}$

To change contrast two reference functions are used. With contrast $\rfloor \geq 0$, i.e., increased contrast, the following function is used:

$$f_{pos}(v) =$$

rac{1.0037575963899724}{1 + exp(6.279 + v cdot 12.558)} - 0.0018787981949862

With contrast $\rfloor < 0$, i.e., decreased contrast, the following function is used:

$$f_{neg}(v) = 0.1755606108304832 \cdot atanh(v \cdot 1.986608 - 0.993304) + 0.5$$

The final value is $v' = (1 - \rfloor) \cdot v + \rfloor \cdot f(v)$.

Brightness and color changes are done via gamma correction.

$$v' = v^{\gamma_{gray} \cdot \gamma_c}$$

with γ_{gray} the gamma for overall lightness and γ_c the per-channel gamma.

Parameters: tensor: image tensor in (N, C, H, W) format gammagrays: tensor of N gamma gray values gammacolors: tensor of $C \cdot N$ gamma values in the format

$$\gamma_{1,1}, \gamma_{1,2}, \cdots, \gamma_{1,C}, \gamma_{2,1}, \gamma_{2,2}, \cdots \gamma_{N,C-1}, \gamma_{N,C}$$

contrasts: tensor of N contrast values in $[-1, 1]$ vmin: minimum lightness value in images vmax: maximum lightness value in images out: output tensor (may be `None`)

Returns: new tensor if `out` is `None`, else `out`

rtype CudaTensor

3.1.4.4 Blur

The following functions apply different types of blur on 2D images. Both input and output must be contiguous and in channel-first format (channel, height, width). All dtypes are supported. Edge values are repeated for locations that fall outside the input image.

Output tensor `out` may be `None`, in which case a new tensor of the same shape and dtype as the input is returned. Output tensor must be same shape and dtype as the input. If output is given `None` is returned.

augpy.**box_blur_single**(*input: augpy._augpy.CudaTensor*, *ksize: int*, *out: augpy._augpy.CudaTensor = None*) → augpy._augpy.CudaTensor

Apply box blur to a single image.

Kernel size describes both width and height in pixels of the area in the input that is averaged for each output pixel. Odd values are recommended for best results. For even values, the center of the kernel is below and to the right of the true center. This means the output is shifted up and left by half a pixel.

Parameters

- **input** (CudaTensor) – image tensor in channel-first format
- **ksize** (*int*) – kernel size in pixels
- **out** (CudaTensor) – output tensor (may be `None`)

Returns new tensor if `out` is `None`, else `out`

Return type *CudaTensor*

augpy.**gaussian_blur_single**(*input: augpy._augpy.CudaTensor*, *sigma: float*, *out: augpy._augpy.CudaTensor = None*) → augpy._augpy.CudaTensor

Apply Gaussian blur to a single image.

Kernel size is calculated like this:

```
ksize = max(3, int(sigma * 6.6 - 2.3) | 1)
```

I.e., `ksize` is at least 3 and always odd.

Parameters

- **input** (CudaTensor) – image tensor in channel-first format
- **sigma** (*float*) – standard deviation of the kernel
- **out** (CudaTensor) – output tensor (may be None)

Returns new tensor if `out` is None, else `out`

Return type *CudaTensor*

augpy.**gaussian_blur**(*input: augpy._augpy.CudaTensor*, *sigmas: augpy._augpy.CudaTensor*, *max_ksize: int*, *out: augpy._augpy.CudaTensor = None*) → augpy._augpy.CudaTensor

Apply Gaussian blur to a batch of images.

Maximum kernel size can be calculated like this:

```
ksize = max(3, int(max(sigmas) * 6.6 - 2.3) | 1)
```

I.e., `ksize` is at least 3 and always odd.

The given kernel size defines the upper limit. The actual kernel size is calculated with the formula above and clipped at the given maximum.

Smaller values can be given to trade speed vs quality. Bigger values typically do not visibly improve quality.

Odd values are strongly recommended for best results. For even values, the center of the kernel is below and to the right of the true center. This means the output is shifted up and left by half a pixel. This can lead to inconsistencies between images in the batch. Images with large sigmas may be shifted, while smaller sigmas mean no shift occurs.

Parameters

- **input** (CudaTensor) – batch tensor with images in first dimension
- **sigmas** (CudaTensor) – float tensor with one sigma value per image in the batch
- **max_ksize** (*int*) – maximum kernel size in pixels
- **out** (CudaTensor) – output tensor (may be None)

Returns new tensor if `out` is None, else `out`

Return type *CudaTensor*

3.1.5 augpy.image

class augpy.image.**DecodeWarp**(*batch_size: int, shape: Tuple[int, int, int], background: augpy._augpy.CudaTensor = None, dtype: augpy._augpy.DLDataType = <augpy._augpy.DLDataType object>, cpu_threads: int = 1, num_buffers: int = 2, decode_buffer_size: Optional[int] = None*)

Bases: object

Use `Decoder.decode()` to decode JPEG images in memory and apply `warp_affine()` into batch tensor buffers.

DecodeWarp instances allocate buffers and decoders on the *current_device*.

Parameters

- **batch_size** (*int*) – number of samples in a batch
- **shape** (*Tuple[int, int, int]*) – shape of one image in the batch (C, H, W)
- **background** (CudaTensor) – tensor with C background color values
- **dtype** (DLDataType) – type used for buffer tensors
- **cpu_threads** (*int*) – number of parallel decoders
- **num_buffers** (*int*) – number of buffers tensors
- **decode_buffer_size** (*Optional[int]*) – size of pre-allocated buffer for decoding; must be larger than the number of subpixels; if None a new buffer is allocated every time

__call__(*batch: dict*) → dict

Decode a list of JPEG images under the `'image'` key and warp them into a batch tensor with the parameters defined by a list of augmentation dicts under key `'augmentation'`.

Each set of augmentation parameters is a dict that contains values for the parameters of the `warp_affine()` function. Additional parameters are ignored.

Parameters **batch** (*dict*) – dict `{'image': [JPEG, JPEG, ...], 'augmentation': [params, params, ...]}`

Returns `batch` where `'image'` is replaced by a batch tensor of transformed images.

Return type dict

finalize_batch(*buffer*)

class augpy.image.**Lighting**(*batch_size: int, channels: int = 3, min_value: Union[int, float] = 0, max_value: Union[int, float] = 255*)

Bases: object

Apply the `lighting()` function to a batch of images. The batch tensor must have format (N, C, H, W), with batch size N, number of channels C, height H, and width W.

Lighting instances allocate buffers on the *current_device*.

Parameters

- **batch_size** (*int*) – number of samples in a batch
- **channels** (*int*) – number of channels C per image
- **min_value** (*Union[int, float]*) – minimum brightness value, typically 0 for 8 bit images

- **max_value** (*Union[int, float]*) – maximum brightness value, typically 255 for 8 bit images

__call__(*batch*)

Apply lighting augmentation to a batch of images in a tensor under the `'image'` key, with parameters parameters defined by a list of augmentation dicts under key `'augmentation'`. Modifies the image tensor in-place.

Each set of augmentation parameters is a dict that contains values for the parameters of the `lighting()` function. Additional parameters are ignored.

Parameters **batch** – dict `{'image': [JPEG, JPEG, ...], 'augmentation': [params, params, ...]}`

Returns `batch` where `'image'` has been modified in-place according to given augmentation parameters.

3.1.6 augpy.numeric_limits

`augpy.numeric_limits.`**CAST_LIMITS**

Dictionary of numeric limits when casting between data types in augpy.

Keys are tuples (d_{from}, d_{to}) for casting from type d_{from} to type d_{to}. Types may be any combination of augpy and numpy types.

Values are tuples (v_{min}, v_{max}), which define the lowest and highest possible value after casting. `None` for either means that all negative or positive d_{from} values are representable by d_{to}.

For casting floating point types to integers these limits may be different from the lowest and highest values of d_{to} respectively. augpy guarantees that, for negative values, the cast value is never lower and for positive values is never higher than the original value. Some values near the end of the value range would otherwise produce inconsistent results.

Example:

```
CAST_LIMITS[numpy.uint16, numpy.uint32] = (None, None)
CAST_LIMITS[augpy.uint64, augpy.int32] = (None, 2147483647)
CAST_LIMITS[numpy.float64, augpy.int16] = (-32768, 32767)
CAST_LIMITS[augpy.float32, numpy.uint64] = (0, 18446744073709551615)
```

CHAPTER

FOUR

C++ REFERENCE

4.1 C++ Reference

4.1.1 Core Functionality

Functionality to catch exceptions, dispatch kernels for different dtypes, and writing kernels.

4.1.1.1 Exceptions

CUDA(*a*)
Wrap a function that returns cudaError_t and throw a *cuda_error* if not `cudaSuccess`.

class `augpy::`**cuda_error** : **private** exception
C++ exception for Cuda errors.

Public Functions

cuda_error(cudaError_t *error*)
Create exception from error code.

const char ***what**() **const noexcept**
Text description of error.

CNMEM(*a*)
Wrap a function that returns *cnmemStatus_t* and throw a *cnmem_error* if not `CNMEM_STATUS_SUCCESS`.

class `augpy::`**cnmem_error** : **private** exception
C++ exception for cnmem errors.

Public Functions

cnmem_error(*cnmemStatus_t error*)
Create exception from error code.

const char ***what**() **const noexcept**
Text description of error.

NVJPEG(*a*)
Wrap a function that returns nvjpegStatus_t and throw a *nvjpeg_error* if not `NVJPEG_STATUS_SUCCESS`.

class `augpy::`**nvjpeg_error** : **private** exception
C++ exception for nvjpeg errors.

Public Functions

nvjpeg_error(nvjpegStatus_t *error*)
Create exception from error code.

const char ***what**() **const noexcept**
Text description of error.

CURAND(*a*)
Wrap a function that returns curandStatus_t and throw a *curand_error* if not `CURAND_STATUS_SUCCESS`.

class `augpy::curand_error` : **private** exception
C++ exception for CuRand errors.

Public Functions

curand_error(curandStatus_t *error*)
Create exception from error code.

const char ***what**() **const noexcept**
Text description of error.

Warning: doxygendefine: Cannot find define "CUBLAS" in doxygen xml output for project "augpy" from directory: /home/docs/checkouts/readthedocs.org/user_builds/augpy/checkouts/latest/doc/source/../xml

Warning: doxygenclass: Cannot find class "augpy::cublas_error" in doxygen xml output for project "augpy" from directory: /home/docs/checkouts/readthedocs.org/user_builds/augpy/checkouts/latest/doc/source/../xml

4.1.1.2 Dispatching

Convenience macros to dispath functions based on tensor data type.

AUGPY_DISPATCH(*TYPE*, *NAME*, ...)
Dispatch all dtypes. Declares tensor dtype as `scalar_t`, signed version of dtype as `sscalar_t`, and `float` (8, 16 bits) `double` (32, 64 bits) as `temp_t`.

Parameters

- `TYPE`: tensor dtype, see *DLDataType*
- `NAME`: name of the dispatched function; used when exceptions are thrown
- `...`: code to execute

AUGPY_DISPATCH_NOEXC(*TYPE*, *NAME*, ...)
Dispatch all dtypes. Does not throw exception when dtype cannot be dispatched. Declares tensor dtype as `scalar_t`, signed version of dtype as `sscalar_t`, and `float` (8, 16 bits) `double` (32, 64 bits) as `temp_t`.

Parameters

- `TYPE`: tensor dtype, see *DLDataType*
- `NAME`: name of the dispatched function; used when exceptions are thrown
- `...`: code to execute

AUGPY_DISPATCH_F (*TYPE*, *NAME*, ...)
Dispatch all dtypes that can use `float` as temp_t. Declares tensor dtype as `scalar_t`, signed version of dtype as `sscalar_t`, and `float` as `temp_t`.

Parameters

- `TYPE`: tensor dtype, see *DLDataType*
- `NAME`: name of the dispatched function; used when exceptions are thrown
- `...`: code to execute

AUGPY_DISPATCH_F_NOEXC (*TYPE*, *NAME*, ...)
Dispatch all dtypes that can use `float` as temp_t. Does not throw exception when dtype cannot be dispatched. Declares tensor dtype as `scalar_t`, signed version of dtype as `sscalar_t`, and `float` as `temp_t`.

Parameters

- `TYPE`: tensor dtype, see *DLDataType*
- `NAME`: name of the dispatched function; used when exceptions are thrown
- `...`: code to execute

AUGPY_DISPATCH_D (*TYPE*, *NAME*, ...)
Dispatch all dtypes that must use `double` as temp_t. Declares tensor dtype as `scalar_t`, signed version of dtype as `sscalar_t`, and `double` as `temp_t`.

Parameters

- `TYPE`: tensor dtype, see *DLDataType*
- `NAME`: name of the dispatched function; used when exceptions are thrown
- `...`: code to execute

AUGPY_DISPATCH_D_NOEXC (*TYPE*, *NAME*, ...)
Dispatch all dtypes that must use `double` as temp_t. Does not throw exception when dtype cannot be dispatched. Declares tensor dtype as `scalar_t`, signed version of dtype as `sscalar_t`, and `double` as `temp_t`.

Parameters

- `TYPE`: tensor dtype, see *DLDataType*
- `NAME`: name of the dispatched function; used when exceptions are thrown
- `...`: code to execute

4.1.1.3 Type Casting

The `saturate_cast` family of function templates provide safe, saturating type casting for both GPU and CPU. Any combination of augpy dtypes can be cast.

Always instantiate the template with both types to ensure that the correct implementation is used.

Cuda type cast intrinsics are used where possible, otherwise `min` and `max` math functions are used. Some type combinations use if-the-else branches on CPU.

template<typename **input_t**, typename **output_t**>

__device__ __host__ __forceinline__ void **saturate_cast**(*input_t vin*, *output_t *vout*)

Cast a `input_t` value to `output_t` and write to `vout` pointer. The cast is done with saturation, meaning that no under or overflow will occur.

It is ensured that, for any casted value pairs (v_{in}, v_{out}) and (v'_{in}, v'_{out}), if $v_{in} \leq (v'_{in}$, then $v_{out} \leq (v'_{out}$. Similarly, if $v_{in} \geq (v'_{in}$, then $v_{out} \geq (v'_{out}$.

When casting from integral to float types, depending on available precision, generally $v_{in} \neq v_{out}$.

The input is rounded to nearest even when casting from float to integral types.

Template Parameters

- **input_t** – input dtype
- **output_t** – output dtype

Parameters

- **vin** – input value to cast
- **vout** – pointer to output value

4.1.1.4 Elementwise Iteration

Warning: doxygenfunction: Cannot find function "elementwise_contiguous_kernel" in doxygen xml output for project "augpy" from directory: /home/docs/checkouts/readthedocs.org/user_builds/augpy/checkouts/latest/doc/source/../xml

template<int **n_tensors**, unsigned int **values_per_thread**, typename **param_t**, typename **F**>
void `augpy::elementwise_kernel`(*array*<tensor_param, *n_tensors*> *tensors*, **const** *param_t constants*, **const** *ndim_array contiguous_strides*, **const** int *ndim*, **const** size_t *count*, size_t *shape0*)

Cuda kernel for elementwise functions on arbitrary tensors. `values_per_thread` defines how many elements in the tensors each thread in each block will calculate. This is done by striding in the first dimension of the tensors. Effectively, `values_per_thread` is never larger than `shape0`.

Parameters

- `tensors`: an array of `n_tensors` tensor_params ; first tensor is output, remainder are inputs; strides must be given in bytes
- `constants`: constant value given to function
- `contiguous_strides`: strides in bytes a contiguous tensor of the same shape would have
- `ndim`: number of dimensions in tensors
- `count`: number of values in tensors
- `values_per_thread`: number of values in tensor each thread in each block computes
- `shape0`: number of elements in first dimension of tensors

Template Parameters

- `n_tensors`: number of tensors given to function
- `param_t`: type of constant value given to kernel function alongside tensors
- `F`: function of type `void F(const array<tensor_param, n_tensors>&, const param_t&)` applied to tensors

template<int **n_tensors**, typename **param_t**, typename **F**>
CudaTensor *augpy::**elementwise_function**(*array*<*CudaTensor**, *n_tensors*> *tensors*, **const** *param_t constant*, unsigned int *blocks_per_sm*, unsigned int *num_threads*, bool *enforce_same_dtype* = true)

Apply a function elementwise to some tensors.

Parameters

- `tensors`: an array of `n_tensors` tensors; first tensor is output, remainder are inputs
- `constant`: constant value given to function
- `blocks_per_sm`: number of blocks to create per SM on the GPU; at least 1
- `num_threads`: number of threads in each block; 0 means auto-select
- `enforce_same_dtype`: if true, throws `std::invalid_argument` if tensors have different dtypes

Template Parameters

- `n_tensors`: number of tensors given to function
- `param_t`: type of constant value given to kernel function alongside tensors
- `F`: function of type `void (*F)(array<tensor_param, n_tensors>, param_t)` applied to tensors

4.1.1.5 Index Translation

template<typename **scalar_t**>
bool augpy::**translate_idx_strided**(*scalar_t* *&*t*, **const** *ndim_array strides*, **const** *ndim_array contiguous_strides*, **const** int *ndim*, **const** size_t *count*, **const** unsigned int *values_per_thread*, size_t &*shape0*)

Translate from contiguous index to a single strided tensor.

Parameters

- `t`: pointer to tensor data
- `strides`: strides in number of elements of the strided tensor
- `contiguous_strides`: strides of the contiguous tensor in bytes
- `ndim`: number of dimensions
- `count`: number of elements in the tensor without the first dimension, i.e., $\frac{numel(t)}{s_0}$
- `values_per_thread`: number of values calculated by each thread
- `shape0`: size of the first dimension; looped over at most values_per_thread times

template<typename **scalar1_t**, typename **scalar2_t**>
bool augpy::**translate_idx_contiguous_strided**(*scalar1_t* *&*t1*, **const** *ndim_array t1_strides*, *scalar2_t* *&*t2*, **const** *ndim_array t2_strides*, **const** int *ndim*, **const** size_t *count*, **const** unsigned int *values_per_thread*, size_t &*shape0*)

Translate from one contiguous to one strided tensor. `t1` must be contiguous, `t2` may be strided.

Parameters

- `t1`: pointer to contiguous tensor data
- `t1_strides`: strides in number of elements of `t1`
- `t2`: pointer to strided tensor data
- `t2_strides`: strides in number of elements of `t2`
- `ndim`: number of dimensions
- `count`: number of elements in the tensor without the first dimension, i.e., $\frac{numel(t)}{s_0}$
- `values_per_thread`: number of values calculated by each thread
- `shape0`: size of the first dimension; looped over at most values_per_thread times

template<typename **scalar1_t**, typename **scalar2_t**>
bool `augpy::translate_idx_strided_strided`(*scalar1_t* *&*t1*, **const** *ndim_array t1_strides*, *scalar2_t* *&*t2*, **const** *ndim_array t2_strides*, **const** *ndim_array contiguous_strides*, **const** int *ndim*, **const** size_t *count*, **const** unsigned int *values_per_thread*, size_t &*shape0*)

Translate contiguous index to two strided tensors. `t1` and `t2` may be strided.

Parameters

- `t1`: pointer to first tensor data
- `t1_strides`: strides in number of elements of `t1`
- `t2`: pointer to second tensor data
- `t2_strides`: strides in number of elements of `t2`
- `contiguous_strides`: strides of the contiguous tensor in bytes
- `ndim`: number of dimensions
- `count`: number of elements in the tensor without the first dimension, i.e., $\frac{numel(t)}{s_0}$
- `values_per_thread`: number of values calculated by each thread
- `shape0`: size of the first dimension; looped over at most values_per_thread times

template<typename **scalar1_t**, typename **scalar2_t**, typename **scalar3_t**>
bool `augpy::translate_idx_strided_strided_strided`(*scalar1_t* *&*t1*, **const** *ndim_array t1_strides*, *scalar2_t* *&*t2*, **const** *ndim_array t2_strides*, *scalar3_t* *&*t3*, **const** *ndim_array t3_strides*, **const** *ndim_array contiguous_strides*, **const** int *ndim*, **const** size_t *count*, **const** unsigned int *values_per_thread*, size_t &*shape0*)

Translate contiguous index to three strided tensors. `t1`, `t2`, and `t3` may be strided.

Parameters

- `t1`: pointer to first tensor data
- `t1_strides`: strides in number of elements of `t1`
- `t2`: pointer to second tensor data
- `t2_strides`: strides in number of elements of `t2`

- `t3`: pointer to third tensor data
- `t3_strides`: strides in number of elements of `t3`
- `contiguous_strides`: strides of the contiguous tensor in bytes
- `ndim`: number of dimensions
- `count`: number of elements in the tensor without the first dimension, i.e., $\frac{numel(t)}{s_0}$
- `values_per_thread`: number of values calculated by each thread
- `shape0`: size of the first dimension; looped over at most values_per_thread times

THREAD_LOOP_1 (*FUN*, *COUNTER*, *P1*, *STRIDE1*)
Loop one strided tensor over first dimension.

Parameters

- `FUN`: code to execute
- `COUNTER`: counter variable, initially set to number of iterations
- `P1`: pointer variable
- `STRIDE1`: stride in number of elements in the first dimension

THREAD_LOOP_2 (*FUN*, *COUNTER*, *P1*, *STRIDE1*, *P2*, *STRIDE2*)
Loop two strided tensors over first dimension.

Parameters

- `FUN`: code to execute
- `COUNTER`: counter variable, initially set to number of iterations
- `P1`: first pointer variable
- `STRIDE1`: first tensor stride in number of elements in the first dimension
- `P2`: second pointer variable
- `STRIDE2`: second tensor stride in number of elements in the first dimension

THREAD_LOOP_3 (*FUN*, *COUNTER*, *P1*, *STRIDE1*, *P2*, *STRIDE2*, *P3*, *STRIDE3*)
Loop three strided tensors over first dimension.

Parameters

- `FUN`: code to execute
- `COUNTER`: counter variable, initially set to number of iterations
- `P1`: first pointer variable
- `STRIDE1`: first tensor stride in number of elements in the first dimension
- `P2`: second pointer variable
- `STRIDE2`: second tensor stride in number of elements in the first dimension
- `P3`: third pointer variable
- `STRIDE3`: third tensor stride in number of elements in the first dimension

4.1.1.6 Info & Profiling Functions

std::tuple<size_t, size_t, size_t> `augpy::meminfo`(int *device_id*)
For the device defined by `device_id`, return the current used, free, and total memory in bytes.

void `augpy::enable_profiler`()
Enable the Cuda profiler.

void `augpy::disable_profiler`()
Disable the Cuda profiler.

nvtxRangeId_t `augpy::nvtx_range_start`(std::string *msg*)
Tell the Nvidia profiler to start a new nvtx range. Can be used to place marks in profiling output.

Return range ID to be used with *nvtx_range_end*

Parameters

- `msg`: Message attached to the range

void `augpy::nvtx_range_end`(nvtxRangeId_t *end*)
Tell the Nvidia profiler to end the given nvtx range.

Parameters

- `end`: ID of the range to end

template<typename **T**, int **N**>
class `augpy::array`
A simple fixed-size array. Importantly `sizeof(array)` gives the actual size of the array in bytes, so it can be used as an argument in function calls etc.

Template Parameters

- `T`: element type
- `N`: length of array

Public Functions

array()
Create a new array, setting all bytes in `x` to zero.

array(*T* **values*, int *n*)
Create a new array, copying `n` values from the given pointer. All remaining bytes in `x` are set to zero.

T &**operator[]**(size_t *idx*)
Return a reference to the element at index idx.

const *T* &**operator[]**(size_t *idx*) **const**
Return a const reference to the element at index idx.

T ***ptr**()
Return a pointer to the internal array storage `x`.

Public Members

T **x**[**N**]
Array that holds the data.

template<class **T**, class ...**Tail**>
array<*T*, 1 + sizeof...(*Tail*)> `augpy::`**make_array**(*T head*, *Tail... tail*)
Similar to std::make_tuple, but makes a fixed-length *array* instead.

Example: `auto tensors = make_array(tensor1, tensor2, tensor3);`

BLOCKS_PER_SM
Default value for blocks to generate per SM. Used to calculate kernel launch config. Currently 8.

4.1.2 Device & Memory

Classes and functions to manage GPU devices and memory.

4.1.2.1 Device Management

augpy gives you fine control over which Cuda device is used and which Cuda stream kernels are run on. All functions are asynchronous by design, so events and streams can be used to synchronize host code.

There are two thread-local global variables that control which device and stream are currently active:

- *current_device*
- *current_stream*

class `augpy::`**CudaEvent**
Convenience wrapper for the cudaEvent_t type.

Public Functions

CudaEvent()
Get a Cuda event from the event pool of the *current_device*.

~CudaEvent () noexcept(false)

cudaEvent_t **get_event**()
Return the wrapped Cuda event.

void **record**()
Record wrapped event on *current_stream*.

bool **query**()
Returns `true` if event has occurred.

void **synchronize**(int *microseconds* = 100)
Block until event has occurred. Checks in `microseconds` interval. Faster intervals make this more accurate, but increase CPU load. Uses standard Cuda busy-waiting method if `microseconds <= 0`.

class `augpy::`**CudaStream**
Convenience wrapper for the cudaStream_t type.

Public Functions

CudaStream(int *device_id* = -1, int *priority* = -1)
Create a new Cuda stream on the given device. Lower numbers mean higher priority, and values are clipped to the valid range. Use *get_device_properties* to get the range of possible values for a device. See cudaStreamCreateWithPriority for more details.

Use `device_id=-1` and `priority=-1` to get the *default_stream*.

CudaStream(cudaStream_t *stream*)
Wrap the given cudaStream_t in a *CudaStream*.

~CudaStream () noexcept(false)

cudaStream_t &**get_stream**()
Return the wrapped Cuda stream.

void **activate**()
Make this the *current_stream* and remember the previous stream.

void **deactivate**()
Make the previous stream the *current_stream*.

void **synchronize**(int *microseconds* = 100)
Block until all work on this stream has finished. Checks in `microseconds` interval. Faster intervals make this more accurate, but increase CPU load. Uses standard Cuda busy-waiting method if `microseconds <= 0`.

std::string **repr**()
Returns a concise string representation of this stream.

int `augpy::`**current_device**
Controls which GPU device is used by each thread.

cudaStream_t `augpy::`**current_stream**
Controls which Cuda stream is used by each thread.

const *CudaStream* `augpy::`**default_stream** = *CudaStream*(-1, -1)
The default Cuda stream as a wrapped *CudaStream*.

4.1.2.2 Device Information

struct `augpy::`**cudaDevicePropEx** : **public** cudaDeviceProp
The cudaDeviceProp struct extended with stream priority fields.

Public Members

int **leastStreamPriority** = 0
Lowest priority a Cuda stream on this device can have.

int **greatestStreamPriority** = 0
Highest priority a Cuda stream on this device can have.

int **coresPerSM** = 0
Number of Cuda cores per SM.

int **numCudaCores** = 0
Total number of Cuda coes.

cudaDevicePropEx `augpy::get_device_properties`(int *device_id*)
Returns the device properties of the given GPU device.

int `augpy::get_num_cuda_cores`(int *device_id*)
Returns the number of Cuda cores of the given GPU device.

int `augpy::cores_per_sm`(int *device_id*)
Given a GPU device id, returns the number of Cuda cores per SM.

int `augpy::cores_per_sm`(int *major*, int *minor*)
Given the major and minor Cuda capability (e.g., 7 and 5), returns the number of Cuda cores per SM.

4.1.2.3 Memory Management

std::tuple<size_t, size_t, size_t> `augpy::meminfo`(int *device_id*)
For the device defined by `device_id`, return the current used, free, and total memory in bytes.

struct `augpy::managed_allocation`
A chunk of managed GPU memory.

Public Functions

managed_allocation(int *device_id*, size_t *size*)
Make new struct filled with `device_id` and `size`. Does not allocate memory. Use *managed_cudamalloc* instead.

void **record**()

Public Members

int **device_id**
GPU device id

size_t **size**
Number of bytes in allocation

void ***ptr**
Pointer to allocated memory

cudaEvent_t **event**
Cuda event used to track whether memory is currently in use

std::shared_ptr<*managed_allocation*> `augpy::managed_cudamalloc`(size_t *size*, int *device_id*)
Malloc `size` bytes ond GPU with given `device id`. Returns *managed_allocation* as std::shared_ptr. Throws *cuda_error*.

One all instances of shared_ptr are deleted, the allocated memory will be marked for deletion/reuse.

void `augpy::managed_cudafree`(void **ptr*)
Frees device memory allocated by *managed_cudamalloc* at the given location. Throws *cuda_error*.

void `augpy::managed_eventalloc`(cudaEvent_t **event*)
Return a Cuda event from the event pool of the *current_device*. Flags `cudaEventBlockingSync` and `cudaEventDisableTiming` are set.

void `augpy::managed_eventfree`(cudaEvent_t *event*)
Mark the given Cuda event as reusable.

void `augpy::init_device`(int *device_id*)
Initialize the GPU with the given `device_id`. You can, but do not need to this manually. It is done for you whenever you request memory or device properties for the first.

void `augpy::release`()
Release all allocated memory on all GPUs. All *CudaTensors* become invalid immediately. Do I have to tell you this is dangerous?

4.1.2.4 cnmem

augpy uses cnmem to manage GPU device memory.

Defines

CNMEM_API

CNMEM_VERSION

Typedefs

typedef struct *cnmemDevice_t_* **cnmemDevice_t**

Enums

enum cnmemStatus_t
Values:

enumerator CNMEM_STATUS_SUCCESS = 0

enumerator CNMEM_STATUS_CUDA_ERROR

enumerator CNMEM_STATUS_INVALID_ARGUMENT

enumerator CNMEM_STATUS_NOT_INITIALIZED

enumerator CNMEM_STATUS_OUT_OF_MEMORY

enumerator CNMEM_STATUS_UNKNOWN_ERROR

enum cnmemManagerFlags_t
Values:

enumerator CNMEM_FLAGS_DEFAULT = 0

enumerator CNMEM_FLAGS_CANNOT_GROW = 1
Default flags.

enumerator CNMEM_FLAGS_CANNOT_STEAL = 2
Prevent the manager from growing its memory consumption.

enumerator CNMEM_FLAGS_MANAGED = 4
Prevent the manager from stealing memory.

Functions

cnmemStatus_t **cnmemInit** (int *numDevices*, **const** *cnmemDevice_t* **devices*, unsigned *flags*)
Initialize the library and allocate memory on the listed devices.

For each device, an internal memory manager is created and the specified amount of memory is allocated (it is the size defined in device[i].size). For each, named stream an additional memory manager is created. Currently, it is implemented as a tree of memory managers: A root manager for the device and a list of children, one for each named stream.

This function must be called before any other function in the library. It has to be called by a single thread since it is not thread-safe.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_INVALID_ARGUMENT, if one of the argument is invalid, CNMEM_STATUS_OUT_OF_MEMORY, if the requested size exceeds the available memory, CNMEM_STATUS_CUDA_ERROR, if an error happens in a CUDA function.

cnmemStatus_t **cnmemFinalize** ()
Release all the allocated memory.

This function must be called by a single thread and after all threads that called cnmemMalloc/cnmemFree have joined. This function is not thread-safe.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called, CNMEM_STATUS_CUDA_ERROR, if an error happens in one of the CUDA functions.

cnmemStatus_t **cnmemRetain** ()
Increase the internal reference counter of the context object.

This function increases the internal reference counter of the library. The purpose of that reference counting mechanism is to give more control to the user over the lifetime of the library. It is useful with scoped memory allocation which may be destroyed in a final memory collection after the end of main(). That function is thread-safe.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called,

cnmemStatus_t **cnmemRelease** ()
Decrease the internal reference counter of the context object.

This function decreases the internal reference counter of the library. The purpose of that reference counting mechanism is to give more control to the user over the lifetime of the library. It is useful with scoped memory allocation which may be destroyed in a final memory collection after the end of main(). That function is thread-safe.

You can use `cnmemRelease` to explicitly finalize the library.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called,

cnmemStatus_t **cnmemRegisterStream** (cudaStream_t *stream*)
Add a new stream to the pool of managed streams on a device.

This function registers a new stream into a device memory manager. It is thread-safe.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_INVALID_ARGUMENT, if one of the argument is invalid,

cnmemStatus_t **cnmemMalloc**(void **ptr*, size_t *size*, cudaStream_t *stream*)

Allocate memory.

This function allocates memory and initializes a pointer to device memory. If no memory is available, it returns a CNMEM_STATUS_OUT_OF_MEMORY error. This function is thread safe.

The behavior of that function is the following:

- If the stream is NULL, the root memory manager is asked to allocate a buffer of device memory. If there's a buffer of size larger or equal to the requested size in the list of free blocks, it is returned. If there's no such buffer but the manager is allowed to grow its memory usage (the CNMEM_FLAGS_CANNOT_GROW flag is not set), the memory manager calls cudaMalloc. If cudaMalloc fails due to no more available memory or the manager is not allowed to grow, the manager attempts to steal memory from one of its children (unless CNMEM_FLAGS_CANNOT_STEAL is set). If that attempt also fails, the manager returns CNMEM_STATUS_OUT_OF_MEMORY.
- If the stream is a named stream, the initial request goes to the memory manager associated with that stream. If a free node is available in the lists of that manager, it is returned. Otherwise, the request is passed to the root node and works as if the request were made on the NULL stream.

The calls to cudaMalloc are potentially costly and may induce GPU synchronizations. Also the mechanism to steal memory from the children induces GPU synchronizations (the manager has to make sure no kernel uses a given buffer before stealing it) and it the execution is sequential (in a multi-threaded context, the code is executed in a critical section inside the cnmem library - no need for the user to wrap cnmemMalloc with locks).

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called, CNMEM_STATUS_INVALID_ARGUMENT, if one of the argument is invalid. For example, ptr == 0, CNMEM_STATUS_OUT_OF_MEMORY, if there is not enough memory available, CNMEM_STATUS_CUDA_ERROR, if an error happens in one of the CUDA functions.

cnmemStatus_t **cnmemFree**(void **ptr*, cudaStream_t *stream*)

Release memory.

This function releases memory and recycles a memory block in the manager. This function is thread safe.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called, CNMEM_STATUS_INVALID_ARGUMENT, if one of the argument is invalid. For example, ptr == 0, CNMEM_STATUS_CUDA_ERROR, if an error happens in one of the CUDA functions.

cnmemStatus_t **cnmemMemGetInfo**(size_t **freeMem*, size_t **totalMem*, cudaStream_t *stream*)

Returns the amount of memory managed by the memory manager associated with a stream.

The pointers totalMem and freeMem must be valid. At the moment, this function has a comple- xity linear in the number of allocated blocks so do not call it in performance critical sections.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called, CNMEM_STATUS_INVALID_ARGUMENT, if one of the argument is invalid, CNMEM_STATUS_CUDA_ERROR, if an error happens in one of the CUDA functions.

cnmemStatus_t **cnmemPrintMemoryState** (FILE **file*, cudaStream_t *stream*)
Print a list of nodes to a file.

This function is intended to be used in case of complex scenarios to help understand the behaviour of the memory managers/application. It is thread safe.

Return CNMEM_STATUS_SUCCESS, if everything goes fine, CNMEM_STATUS_NOT_INITIALIZED, if the *cnmemInit* function has not been called, CNMEM_STATUS_INVALID_ARGUMENT, if one of the argument is invalid. For example, used_mem == 0 or free_mem == 0, CNMEM_STATUS_CUDA_ERROR, if an error happens in one of the CUDA functions.

const char ***cnmemGetErrorString** (*cnmemStatus_t status*)
Converts a cnmemStatus_t value to a string.

struct cnmemDevice_t_
#include <cnmem.h>

Public Members

int **device**
The device number.

size_t **size**
The size to allocate for that device. If 0, the implementation chooses the size.

int **numStreams**
The number of named streams associated with the device. The NULL stream is not counted.

cudaStream_t ***streams**
The streams associated with the device. It can be NULL. The NULL stream is managed.

size_t ***streamSizes**
The size reserved for each streams. It can be 0.

4.1.3 Tensors

augpy's `CudaTensor` class is a backward compatible extension to the *DLPack* specification. This allows trivial conversion to and from DLPack tensors and thus exchange of tensors between frameworks.

Currently, only GPU tensors are supported.

4.1.3.1 Data types

`CudaTensors` can have the following data types, defined as `DLDataType`.

Note: Only scalar data types are supported, so lanes is always 1.

const *DLDataType* `augpy::`**dldtype_int8** = {kDLInt, 8, 1}
8 bit signed integer.

const *DLDataType* `augpy::`**dldtype_uint8** = {kDLUInt, 8, 1}
8 bit unsigned integer.

const *DLDataType* `augpy::`**dldtype_int16** = {kDLInt, 16, 1}
16 bit signed integer.

const *DLDataType* `augpy::dldtype_uint16` = {kDLUInt, 16, 1}
16 bit unsigned integer.

const *DLDataType* `augpy::dldtype_int32` = {kDLInt, 32, 1}
32 bit signed integer.

const *DLDataType* `augpy::dldtype_uint32` = {kDLUInt, 32, 1}
32 bit unsigned integer.

const *DLDataType* `augpy::dldtype_int64` = {kDLInt, 64, 1}
64 bit signed integer.

const *DLDataType* `augpy::dldtype_uint64` = {kDLUInt, 64, 1}
64 bit unsigned integer.

const *DLDataType* `augpy::dldtype_float16` = {kDLFloat, 16, 1}
16 bit (half precision) float.

Note not yet supported

const *DLDataType* `augpy::dldtype_float32` = {kDLFloat, 32, 1}
32 bit (single precision) float.

const *DLDataType* `augpy::dldtype_float64` = {kDLFloat, 64, 1}
64 bit (double precision) float.

template<typename **scalar_t**>
DLDataType `augpy::get_dldatatype()`
Returns the corresponding *DLDataType* for type `scalar_t`

Template Parameters

- `scalar_t`: input type

bool `augpy::dldatatype_equals(`*DLDataType t1*, *DLDataType t2*`)`
Returns `true` if both data types are the same.

4.1.3.2 CudaTensor

DLTENSOR_MAX_NDIM
Maximum number of dimensions a *CudaTensor* can have. Currently 6.

struct `augpy::CudaTensor` : **public** *DLManagedTensor*
Augpy's tensor class. It is a backwards compatible extension to the DLPack. specification.

See *DLPack* for the full documentation.

It supports all the usual operations you would expect from a full-featured tensor class, like complex indexing and slicing.

Copy, math, and comparison operations are provided as separate functions to call on tensors.

Public Functions

CudaTensor(int64_t *shape, int ndim, DLDataType dtype, int device_id)
Create a new tensor with the given shape, dtype, on a specific device.

Parameters

- `shape`: Pointer to a shape array
- `ndim`: number of dimensions, i.e., length of the `shape` array
- `dtype`: data type of the new tensor
- `device_id`: Cuda GPU device id where tensor memory is allocated

CudaTensor(std::vector<int64_t> shape, DLDataType dtype, int device_id)
Alias for *CudaTensor(int64_t*, int, DLDataType, int)* called with `shape.data()` and `shape.size()`.

CudaTensor(*CudaTensor* *parent, int ndim, int64_t *shape)
Create a new tensor that borrows memory from a parent tensor, but has a different shape

Parameters

- `parent`: Parent tensor to borrow memory from
- `ndim`: number of dimensions of new tensor
- `shape`: shape of new tensor, array of length `ndim`

CudaTensor(*CudaTensor* *parent, int ndim, int64_t *shape, int64_t *strides, int64_t byte_offset)
Create a new tensor that borrows memory from a parent tensor, but has a different shape, may stride, and start at a different offset.

Parameters

- `parent`: Parent tensor to borrow memory from
- `ndim`: number of dimensions of new tensor
- `shape`: shape of new tensor, array of length `ndim`
- `strides`: stride distances of the tensor, array of length `ndim`
- `byte_offset`: start position in parent memory in bytes

CudaTensor(*CudaTensor* *parent)
Create an exact copy of the `parent` tensor, borrowing its memory.

CudaTensor(*DLManagedTensor* *parent)
Wrap a *DLManagedTensor* inside a *CudaTensor*, borrowing its memory.

~CudaTensor () noexcept(false)
Delete this *CudaTensor*. Calls the *DLManagedTensor::deleter* function if *DLManagedTensor::manager_ctx* is also set.

The *managed_allocation* will be marked as orphaned/ready for reuse if this tensor is the last remaining tensor that references it.

void ***ptr**()
Return a pointer to the first element in this tensor. Resolves *DLTensor::byte_offset*.

void **record**()
Mark this tensor to be in use by calling *CudaEvent::record* on its event.

cudaEvent_t **get_event**()
Return the Cuda event used to record

bool **is_contiguous**()
Returns `true` if the tensor is contiguous, i.e., elements are located next to each other in memory and in dimensions are not reversed.

CudaTensor ***index**(ssize_t *i*)
Index this tensor in the first dimension at index `i`. Behaves like numpy indexing, i.e, index from the back if `i` is negative where `-1` refers to the last element.

CudaTensor ***slice_simple**(py::slice *slice*)
Slice this tensor in the first dimension. Behaves like numpy slicing, i.e, start, stop, and step may be negative.

CudaTensor ***slice_complex**(py::tuple *slices*)
Slice this tensor in up to *DLTensor::ndim* dimensions. Behaves like numpy slicing, i.e, start, stop, and step may be negative.

void **setitem_index**(ssize_t *index*, *CudaTensor* **src*)
Read items from `src` and write them into the tensor at positions referenced by an `index`.

void **setitem_simple**(py::slice *slice*, *CudaTensor* **src*)
Read items from `src` and write them into this tensor at positions referenced by a `slice`.

void **setitem_complex**(py::tuple *slices*, *CudaTensor* **src*)
Read items from `src` and write them into this tensor at positions referenced by a number of `slices`.

CudaTensor ***fill_index**(ssize_t *index*, double *scalar*)
Fill this tensor with the given `scalar` value at positions referenced by an `index`. Supports broadcasting.

CudaTensor ***fill_simple**(py::slice *slice*, double *scalar*)
Fill this tensor with the given `scalar` value at positions referenced by a `slice`. Supports broadcasting.

CudaTensor ***fill_complex**(py::tuple *slices*, double *scalar*)
Fill this tensor with the given `scalar` value at positions referenced by a number of `slices`. Supports broadcasting.

CudaTensor ***reshape**(std::vector<int64_t> *shape*)
Returns a new tensor with the given shape that borrows memory from this tensor. Number of elements cannot change and this tensor must be contiguous.

std::string **repr**()
Returns a string representation of this tensor, e.g., `<CudaTensor shape=(1, 2, 3), device=0, dtype=uint8>`.

py::tuple **pyshape**()
Returns the shape of this tensor as a Python tuple.

py::tuple **pystrides**()
Returns the strides of this tensor as a Python tuple.

CudaTensor *`augpy::copy`(*CudaTensor* **src*, *CudaTensor* **dst*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)
Copy `dst` into `dst`. Supports broadcasting.

Return `dst`

CudaTensor *augpy::**fill**(double *scalar*, *CudaTensor* *dst*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)
Fill `dst` with the given `scalar` value.

Return `dst`

CudaTensor *augpy::**cast_tensor**(*CudaTensor* *tensor*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)
Read values from `tensor`, cast them to the data type of `out` and store them there. `tensor` and `out` must have the same shape.

CudaTensor *augpy::**cast_type**(*CudaTensor* *tensor*, *DLDataType* *dtype*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)
Create a new tensor with values from `tensor` cast to the given data type `dtype`.

CudaTensor *augpy::**empty_like**(*CudaTensor* *tensor*)
Create a new empty tensor with the same shape and dtype on the same device.

typedef *array*<int64_t, DLTENSOR_MAX_NDIM> augpy::**ndim_array**
`int64` array of length *DLTENSOR_MAX_NDIM*. Used to store shape or strides.

4.1.3.3 Tensor Math

For these functions, the `result` parameter is optional. If `result` is `NULL` a new tensor of appropriate size is created and returned. If `result` is not `NULL`, use the given tensor as output and return `NULL`.

For basic math functions all inputs and the `result` tensor must have the same data type.

For comparison functions `uint8` is used as result. A value of `1` means the condition is fulfilled, otherwise it is `0`.

Unless otherwise stated, all functions support all data type, broadcasting, and work with strided tensors.

The `blocks_per_sm` and `num_threads` control the kernel launch parameters. The defaults are probably fine, but they can be used to get some more speed if you optimize for specific hardware.

CudaTensor *augpy::**add_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)
Add a `scalar` value to a `tensor`.

CudaTensor *augpy::**add_tensor**(*CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)
Add `tensor2` to `tensor1`.

CudaTensor *augpy::**sub_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)
Subtract a `scalar` value from a `tensor`.

CudaTensor *augpy::**rsub_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)
Subtract a `tensor` from a `scalar` value.

CudaTensor *augpy::**sub_tensor**(*CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)
Subtract `tensor2` from `tensor1`.

CudaTensor *augpy::**mul_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

Multiply a `tensor` by a `scalar` value.

CudaTensor *augpy::**mul_tensor**(*CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

Multiply `tensor1` by `tensor2`.

CudaTensor *augpy::**div_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

Divide a `tensor` by a `scalar` value.

CudaTensor *augpy::**rdiv_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

Divide a `scalar` value by a `tensor`.

CudaTensor *augpy::**div_tensor**(*CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

Divide `tensor1` by `tensor2`.

CudaTensor *augpy::**fma**(double *scalar*, *CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)

Compute a fused multiply-add on a scalar and two tensors, i.e., $r = s \cdot t_1 \cdot t_2$.

If `tensor1` has an unsigned integer data type, then `tensor2` must have the signed version of the same type, e.g., a `uint8` tensor must be paired with a `int8` tensor.

CudaTensor *augpy::**gemm**(*CudaTensor* *A*, *CudaTensor* *B*, *CudaTensor* *C*, double *alpha*, double *beta*)

Uses CuBLAS to Calculate the matrix multiplication of two 2D tensors. More specifically calculates

$$C = A \times (\alpha \cdot B) + \beta \cdot C$$

Only `float` and `double` data types are supported and all tensors must have the same data type. All tensors must be contiguous.

Returns a new tensor if `C` is `NULL`, otherwise `C` is returned.

CudaTensor *augpy::**lt_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

`tensor < scalar`.

CudaTensor *augpy::**lt_tensor**(*CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

`tensor1 < tensor2`.

CudaTensor *augpy::**le_scalar**(*CudaTensor* *tensor*, double *scalar*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

`tensor <= scalar`.

CudaTensor *augpy::**le_tensor**(*CudaTensor* *tensor1*, *CudaTensor* *tensor2*, *CudaTensor* *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

`tensor1 <= tensor2`.

CudaTensor *augpy::**gt_scalar**(*CudaTensor *tensor*, double *scalar*, *CudaTensor *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

tensor > scalar.

CudaTensor *augpy::**ge_scalar**(*CudaTensor *tensor*, double *scalar*, *CudaTensor *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

tensor >= scalar.

CudaTensor *augpy::**eq_scalar**(*CudaTensor *tensor*, double *scalar*, *CudaTensor *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

tensor == scalar.

CudaTensor *augpy::**eq_tensor**(*CudaTensor *tensor1*, *CudaTensor *tensor2*, *CudaTensor *out*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)

tensor1 == tensor2.

4.1.3.4 Tensor Management

Converting from and to arrays or tensors, exporting augpy's *CudaTensors* to other frameworks, and importing existing tensors from other frameworks without copying.

py::array *augpy::**tensor_to_array1**(*CudaTensor *tensor*)

Copy a given tensor to a new numpy array. This initiates an asynchronous copy from device to host memory.

py::array *augpy::**tensor_to_array2**(*CudaTensor *tensor*, py::buffer *array*)

Copy a given tensor to a numpy array created from the given buffer `array`. This initiates an asynchronous copy from device to host memory.

CudaTensor *augpy::**array_to_tensor1**(py::buffer *array*, int *device_id*)

Copy a Python buffer into a new tensor on the specified GPU device. This initiates an asynchronous copy from host to device memory.

CudaTensor *augpy::**array_to_tensor2**(py::buffer *array*, *CudaTensor *tensor*)

Copy a Python buffer to a tensor created from the given buffer `tensor`. This initiates an asynchronous copy from host to device memory.

CudaTensor *augpy::**import_dltensor**(py::capsule *tensor_capsule*, **const** char *name*)

Import a GPU tensor from another library into augpy.

Note This requires explicit synchronization if augpy or the interfacing library is running operations on streams other than the *default_stream*.

Parameters

- `tensor_capsule`: a Python capsule object that contains a *DLManagedTensor*
- `name`: name under which the tensor is stored in the capsule, e.g., `"dltensor"` for Pytorch

py::capsule *augpy::**export_dltensor**(py::object *pytensor*, std::string *name*, bool *destruct*)

Export a GPU tensor to be used by another library.

Note This requires explicit synchronization if augpy or the interfacing library is running operations on streams other than the *default_stream*.

Parameters

- `pytensor`: Python-wrapped *CudaTensor*

- `name`: name under which the tensor is stored in the returned capsule, e.g., `"dltensor"` for Pytorch
- `destruct`: if `true`, add a destructor to the capsule which will delete the tensor when the capsule is deleted; only set to `false` if you know what you're doing

4.1.3.5 Utility Functions

Functions that help writing functions that operate on tensors.

bool `augpy::array_equals`(int *dim0*, int *ndim*, int64_t **array1*, int64_t **array2*)
Returns `true` if `array1[dim] == array2[dim]` for all dimensions from `dim0` to `ndim-1`.

void `augpy::assert_contiguous`(*CudaTensor* **t*)
Throws `std::invalid_argument` if `t` is `NULL` or not contiguous.

size_t `augpy::numel`(*CudaTensor* **tensor*)
Returns the number of elements in the tensor.

size_t `augpy::numel`(*DLTensor* **tensor*)
Returns the number of elements in the tensor.

size_t `augpy::numel`(*DLTensor* &*tensor*)
Returns the number of elements in the tensor.

size_t `augpy::numel`(py::buffer_info &*array*)
Returns the number of elements in the array.

template<typename **scalar_t**>
size_t `augpy::numel`(*scalar_t* **shape*, size_t *ndim*)
Returns the number of elements in the tensor with the given shape.

template<typename **scalar_t**>
size_t `augpy::numel`(std::vector<*scalar_t*> &*shape*)
Returns the number of elements in the tensor with the given shape.

size_t `augpy::numbytes`(*CudaTensor* **tensor*)
Returns the number of bytes occupied by this tensor.

size_t `augpy::numbytes`(*DLTensor* **tensor*)
Returns the number of bytes occupied by this tensor.

size_t `augpy::numbytes`(py::buffer_info &*array*)
Returns the number of bytes occupied by this tensor.

bool `augpy::check_contiguous`(*CudaTensor* **tensor*)
Returns `true` if the tensor is contiguous.

bool `augpy::check_contiguous`(*DLTensor* **tensor*)
Returns `true` if the tensor is contiguous.

bool `augpy::check_contiguous`(py::buffer_info &*array*)
Returns `true` if the array is contiguous.

void `augpy::check_tensor`(*CudaTensor* **tensor*, size_t *min_size*, bool *contiguous*)
Check whether tensor is not `NULL`, has at least a minimum size in bytes, and is contiguous.

Parameters

- `tensor`: tensor to check
- `min_size`: check whether `numbytes(tensor) >= min_size`
- `contiguous`: if `true`, check whether tensor is contiguous and return `false` if not

void `augpy::check_same_device`(*DLTensor t1*, *DLTensor t2*)
Check whether `t1` and `t2` are located on the same GPU device. If not, raise `std::invalid_argument`.

void `augpy::check_same_dtype_device`(*DLTensor t1*, *DLTensor t2*)
Check whether `t1` and `t2` have the same dtype and are located on the same GPU device. If not, raise `std::invalid_argument`.

void `augpy::check_same_dtype_device_shape`(*DLTensor t1*, *DLTensor t2*)
Check whether `t1` and `t2` have the same dtype, are located on the same GPU device, and have the same shape. If not, raise `std::invalid_argument`.

void `augpy::calc_threads`(unsigned int &*threads*, int *device_id*)
If `threads == 0`, set `threads` to *cores_per_sm(device_id)*.

void `augpy::calc_blocks_values_1d`(*DLTensor t*, unsigned int &*num_blocks*, size_t &*num*, unsigned int &*values_per_thread*, unsigned int *threads*, unsigned int *blocks_per_sm*)
Use heuristics to calculate how many blocks and values per thread to use for the given 1D tensor.

Values per thread v is calculated based on the number of elements in the tensor `t`, the number of SMs on the device N_{sm}, the number of blocks per sm B_{sm}, and the number of threads per block N_t:

$$v = \left\lceil \frac{numel(t)}{N_{sm} \cdot B_{sm} \cdot N_t} \right\rceil$$

The number of blocks B is then calculated like this:

$$B = \left\lceil \frac{\lceil numel(t)/v \rceil}{N_t} \right\rceil$$

Parameters

- `t`: input tensor to operate on
- `num_blocks`: output value, number of blocks in the grid
- `num`: output value, number of elements in t, i.e., `numel(t)`
- `values_per_thread`: input/output value, if `>0` specifies the values per thread to use, otherwise will hold the calculated value
- `threads`: number of threads in each block
- `blocks_per_sm`: how many blocks to generate per SM on the device; defaults to *BLOCKS_PER_SM*

void `augpy::calc_blocks_values_nd`(*DLTensor t*, dim3 &*grid*, size_t &*count*, unsigned int &*values_per_thread*, unsigned int *threads*, unsigned int *blocks_per_sm*)
Similar to *calc_blocks_values_1d*, but for ND tensors. Use heuristics to calculate the size of the block grid and values per thread to use for the given ND tensor.

For ND tensors, values per thread only applies to the first dimension. The same heuristics are used, but v cannot exceed the size of the first dimension s_0, so $v' = min(v, s_0)$. `grid.x` is therefore $\left\lceil \frac{s_0}{v'} \right\rceil$ and `grid.y` is $\left\lceil \frac{numel(t)}{s_0 \cdot N_t} \right\rceil$.

Parameters

- `t`: input tensor to operate on
- `grid`: output value, the block grid used for the kernel launch; `grid.x` will hold the number of iterations in the first dimension, `grid.y` the number of blocks required for the remaining dimensions
- `count`: output value, number of elements in t starting with the second dimension, i.e., `numel(t) / t.shape[0]`
- `values_per_thread`: input/output value, if `>0` specifies the values per thread to use, otherwise will hold the calculated value
- `threads`: number of threads in each block
- `blocks_per_sm`: how many blocks to generate per SM on the device; defaults to *BLOCKS_PER_SM*

void `augpy::calculate_contiguous_strides`(*DLTensor t*, *ndim_array &contiguous_strides*)

Calculate the strides in number of elements the given tensor `t` would have if it was contiguous.

bool `augpy::calculate_broadcast_strides`(*DLTensor t_src*, *DLTensor t_dst*, *ndim_array &src_strides*, **const** int *t_src_index*)

If possible, calculate the strides that are needed to broadcast `t_src` to `t_dst`.

Broadcasting is possible if `t_src.ndim <= t_dst.ndim` and every dimension up to `t_src.ndim` is either the same or `t_src` shape is 1. Stride in a broadcastable dimensions is zero.

Return `true` if broadcasting was used

Parameters

- `t_src`: source tensor to broadcast
- `t_dst`: target tensor to broadcast to
- `src_strides`: output value, strides required to broadcast
- `t_src_index`: only used for error message formatting, index of `t_src` in the function call

Exceptions

- `std::invalid_argument`: if broadcasting not possible

bool `augpy::calculate_broadcast_output_shape`(*DLTensor t1*, *DLTensor t2*, int &*ndim*, int64_t **shape*)

If possible, calculate the output shape when broadcasting tensors `t1` and `t2` together. Output shape will have `max(t1.ndim, t2.ndim)` dimensions. To broadcast, both tensors must either match the size of a dimension, or one of them must have size 1. The other tensor then determines the size of the dimension.

Parameters

- `t1`: first tensor
- `t2`: second tensor
- `ndim`: output value, number of dimensions in output
- `shape`: output value, shape array, must have at least length `max(t1.ndim, t2.ndim)`

bool `augpy::calculate_broadcast_output_shape`(*DLTensor t1*, *DLTensor t2*, int &*ndim*, *ndim_array &shape*)

Alias for *calculate_broadcast_output_shape(DLTensor, DLTensor, int&, int64_t*)*.

CudaTensor *`augpy::create_output_tensor`(*CudaTensor* **tensors*, int *n_tensors*, bool *allow_null*)
For all tensors in the given array, get the maximum size in each dimension and create a new tensor of that shape.

Does not check whether tensors are broadcastable.

Parameters

- `tensors`: tensors that will be broadcast to output
- `n_tensors`: number of tensors in array
- `allow_null`: if `true`, allow tensors to be `NULL`, otherwise throw `std::invalid_argument`

void `augpy::coalesce_dimensions`(std::vector<*DLTensor*> &*tensors*)
Manipulate the shapes and strides of the given tensors to coalesce (remove) unnecessary dimensions, thus simplifying the tensors.

A dimension can be coalesced if all tensors are either contiguous in that dimension or have less dimensions.

Warning Output is only valid if tensors have strides produced by *calculate_broadcast_strides*. Tensors must either have the same shape, or be broadcast and thus appear non-contiguous.

4.1.3.6 DLPack

This is the documentation of the common DLPack header. To quote the readme:

DLPack is an open in-memory tensor structure to for sharing tensor among frameworks. DLPack enables

- Easier sharing of operators between deep learning frameworks.
- Easier wrapping of vendor level operator implementations, allowing collaboration when introducing new devices/ops.
- Quick swapping of backend implementations, like different version of BLAS
- For final users, this could bring more operators, and possibility of mixing usage between frameworks.

Please refer to their Github for more details.

The common header of DLPack.

Copyright (c) 2017 by Contributors

Defines

`DLPACK_EXTERN_C`

`DLPACK_VERSION`
The current version of dlpack.

`DLPACK_DLL`
DLPACK_DLL prefix for windows.

Typedefs

typedef struct *DLManagedTensor* **DLManagedTensor**
C Tensor object, manage memory of *DLTensor*. This data structure is intended to faciliate the borrowing of *DLTensor* by another framework. It is not meant to transfer the tensor. When the borrowing framework doesn't need the tensor, it should call the deleter to notify the host that the resource is no longer needed.

Enums

enum DLDeviceType
The device type in *DLContext*.

Values:

enumerator kDLCPU = 1
CPU device.

enumerator kDLGPU = 2
CUDA GPU device.

enumerator kDLCPUPinned = 3
Pinned CUDA GPU device by cudaMallocHost.

Note kDLCPUPinned = kDLCPU | kDLGPU

enumerator kDLOpenCL = 4
OpenCL devices.

enumerator kDLVulkan = 7
Vulkan buffer for next generation graphics.

enumerator kDLMetal = 8
Metal for Apple GPU.

enumerator kDLVPI = 9
Verilog simulator buffer.

enumerator kDLROCM = 10
ROCm GPUs for AMD GPUs.

enumerator kDLExtDev = 12
Reserved extension device type, used for quickly test extension device The semantics can differ depending on the implementation.

enum DLDataTypeCode
The type code options *DLDataType*.

Values:

enumerator kDLInt = 0U

enumerator kDLUInt = 1U

enumerator kDLFloat = 2U

struct DLContext
#include <*dlpack.h*> A Device context for Tensor and operator.

Public Members

DLDeviceType **device_type**
The device type used in the device.

int **device_id**
The device index.

struct DLDataType
#include <*dlpack.h*> The data type the tensor can hold.

Examples

- float: type_code = 2, bits = 32, lanes=1
- float4(vectorized 4 float): type_code = 2, bits = 32, lanes=4
- int8: type_code = 0, bits = 8, lanes=1

Public Members

uint8_t **code**
Type code of base types. We keep it uint8_t instead of DLDataTypeCode for minimal memory footprint, but the value should be one of DLDataTypeCode enum values.

uint8_t **bits**
Number of bits, common choices are 8, 16, 32.

uint16_t **lanes**
Number of lanes in the type, used for vector types.

struct DLTensor
#include <*dlpack.h*> Plain C Tensor object, does not manage memory.

Public Members

void *__data__
The opaque data pointer points to the allocated data. This will be CUDA device pointer or cl_mem handle in OpenCL. This pointer is always aligns to 256 bytes as in CUDA.

For given *DLTensor*, the size of memory required to store the contents of data is calculated as follows:

```
static inline size_t GetDataSize(const DLTensor* t) {
  size_t size = 1;
  for (tvm_index_t i = 0; i < t->ndim; ++i) {
    size *= t->shape[i];
  }
  size *= (t->dtype.bits * t->dtype.lanes + 7) / 8;
  return size;
}
```

DLContext **ctx**
The device context of the tensor.

int **ndim**
Number of dimensions.

DLDataType **dtype**
The data type of the pointer.

int64_t ***shape**
The shape of the tensor.

int64_t ***strides**
strides of the tensor (in number of elements, not bytes) can be NULL, indicating tensor is compact and row-majored.

uint64_t **byte_offset**
The offset in bytes to the beginning pointer to data.

struct DLManagedTensor
#include <dlpack.h> C Tensor object, manage memory of *DLTensor*. This data structure is intended to faciliate the borrowing of *DLTensor* by another framework. It is not meant to transfer the tensor. When the borrowing framework doesn't need the tensor, it should call the deleter to notify the host that the resource is no longer needed.

Subclassed by *augpy::CudaTensor*

Public Members

DLTensor **dl_tensor**
DLTensor which is being memory managed.

void ***manager_ctx**
the context of the original host framework of *DLManagedTensor* in which *DLManagedTensor* is used in the framework. It can also be NULL.

void (***deleter**)(**struct** *DLManagedTensor* *self)
Destructor signature void (*)(void*) - this should be called to destruct manager_ctx which holds the *DLManagedTensor*. It can be NULL if there is no way for the caller to provide a reasonable destructor. The destructors deletes the argument self as well.

4.1.4 Reductions

Reduction operations like sums on tensors.

CudaTensor *augpy::**sum**(*CudaTensor* **tensor*, bool *upcast*)
Sum all elements in a tensor with saturation.

Return sum value as scalar tensor

Parameters

- `tensor`: tensor to sum, must be contiguous
- `upcast`: if `true`, returns temp dtype

CudaTensor *augpy::**sum_axis**(*CudaTensor* **tensor*, int *axis*, bool *keepdim* = false, bool *upcast* = false, *CudaTensor* **out* = nullptr, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 512)
Sum of all elements along an axis in a tensor with saturation.

Return new tensor with values summed along axis if `out` is `NULL`, else `out`

Parameters

- `tensor`: tensor to sum, may be strided
- `axis`: axis index to sum along

- `keepdim`: if `true`, keep sum axis dimension with length 1
- `upcast`: if `true`, returns tensor with temp dtype
- `out`: output tensor (may be `NULL`), must have correct shape
- `blocks_per_sm`:
- `num_threads`:

CudaTensor *`augpy::all`(*CudaTensor *tensor*)
Check whether all elements in a tensor are greater zero.

Return `0` or `1` as scalar `uint8` tensor

Parameters

- `tensor`: tensor to sum, must be contiguous

4.1.5 Random Number Generation

Classes for random number generation. augpy's speciality is that all dtypes are supported for all distributions. For example, it is possible to fill an integer tensor with approximately Gaussian distributed numbers.

using `augpy::rng_t` = curandStateXORWOW_t

class `augpy::RandomNumberGenerator`
A convenient wrapper for cuRAND methods that fill tensors with pseudo-random numbers.

Public Functions

`RandomNumberGenerator`(py::object **device_id*, py::object **seed*)
Create a new RNG instance. `device_id` and `seed` may both be Python `None`. The behavior is identical to *RandomNumberGenerator(int*, unsigned long long*)* with `NULL` pointers.

`RandomNumberGenerator`(int **device_id*, unsigned long long **seed*)
Create a new RNG instance.

Parameters

- `device_id`: GPU device ID; if `NULL`, *current_device* is used
- `seed`: random seed; if `NULL`, read values from std::random_device to create a random seed.

void **`uniform`**(*CudaTensor* **target*, double *vmin*, double *vmax*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)
Fill `target` tensor with uniformly distributed numbers in $[v_{min}, v_{max})$.

Note This is supported for integer tensors. Values are cast from float or double down to the integer type. The mean of the values is approximately $\frac{v_{max}+v_{min}}{2}$.

Warning Saturation is not used. v_{min} and v_{max} must be representable in the target tensor data type.

Parameters

- `target`: tensor to fill
- `vmin`: minimum value; can occur
- `vmax`: maximum value; does not occur

- `blocks_per_sm`: number of blocks per SM
- `num_threads`: number of threads per block See *Blocks and threads*

void **gaussian**(*CudaTensor* *target, double *mean*, double *std*, unsigned int *blocks_per_sm* = BLOCKS_PER_SM, unsigned int *num_threads* = 0)

Fill `target` tensor with Gaussian distributed numbers with specified `mean` and standard deviation `std`.

Note This is supported for integer tensors. Values are drawn from the given distribution, then rounded and cast to the data type of the tensor with saturation. The values in an integer tensor are thus only approximately Gaussian distributed.

Parameters

- `target`: tensor to fill
- `mean`: Gaussian mean
- `std`: Gaussian standard deviation
- `blocks_per_sm`: number of blocks per SM
- `num_threads`: number of threads per block See *Blocks and threads*

4.1.6 Image Functions

4.1.6.1 JPEG Decoding

Hybrid JPEG decoding on CPU and GPU using Nvjpeg.

class `augpy::Decoder`

Wrapper for Nvjpeg-based JPEG decoding.

Public Functions

Decoder(size_t *device_padding*, size_t *host_padding*, bool *gpu_huffman*)

Create a new decoder on the *current_device*.

Parameters

- `device_padding`: see Nvjpeg docs
- `host_padding`: see Nvjpeg docs
- `gpu_huffman`: enable Huffman decoding on the GPU; not recommended unless you really need to offload from CPU

~Decoder()

CudaTensor ***decode**(std::string *data*, *CudaTensor* **buffer*)

Decode a JPEG image using Nvjpeg. Output is in (H, W, C) format and resides on the GPU device.

Return decoded image on GPU in (H, W, C) format

Parameters

- `data`: compressed JPEG image as a JFIF string, i.e., the full file contents
- `buffer`: optional buffer to use; may be `NULL`; if not `NULL` must be big enough to contain the decoded image

4.1.6.2 Blur

The following functions apply different types of blur on 2D images. Both input and output must be contiguous and in channel-first format (channel, height, width). All dtypes are supported. Edge values are repeated for locations that fall outside the input image.

Output tensor `out` may be `NULL`, in which case a new tensor of the same shape and dtype as the input is returned. Output tensor must be same shape and dtype as the input. If output is given `NULL` is returned.

CudaTensor *`augpy::box_blur_single`(*CudaTensor *input*, int *ksize*, *CudaTensor* **out*)

Apply box blur to a single image.

Kernel size describes both width and height in pixels of the area in the input that is averaged for each output pixel. Odd values are recommended for best results. For even values, the center of the kernel is below and to the right of the true center. This means the output is shifted up and left by half a pixel.

Return new tensor if `out` is `NULL`, else `out`

Parameters

- `input`: image tensor in (C, H, W) format.
- `ksize`: kernel size in pixels
- `out`: output tensor (may be `NULL`)

CudaTensor *`augpy::gaussian_blur_single`(*CudaTensor *input*, float *sigma*, *CudaTensor* **out*)

Apply Gaussian blur to a single image.

Kernel size is calculated like this:

```
ksize = max(3, int(sigma * 6.6 - 2.3) | 1)
```

I.e., `ksize` is at least 3 and always odd.

Return new tensor if `out` is `NULL`, else `out`

Parameters

- `input`: image tensor in (N, C, H, W) format
- `sigma`: standard deviation of the kernel
- `out`: output tensor (may be `NULL`)

CudaTensor *`augpy::gaussian_blur`(*CudaTensor *input*, *CudaTensor* **sigmas*, int *max_ksize*, *CudaTensor* **out*)

Apply Gaussian blur to a batch of images.

Maximum kernel size can be calculated like this: `ksize = max(3, int(max(sigmas) * 6.6 - 2.3) | 1)`

I.e., `ksize` is at least 3 and always odd.

The given kernel size defines the upper limit. The actual kernel size is calculated with the formula above and clipped at the given maximum.

Smaller values can be given to trade speed vs quality. Bigger values typically do not visibly improve quality.

Odd values are strongly recommended for best results. For even values, the center of the kernel is below and to the right of the true center. This means the output is shifted up and left by half a pixel. This can lead to inconsistencies between images in the batch. Images with large sigmas may be shifted, while smaller sigmas mean no shift occurs.

Return new tensor if `out` is `NULL`, else `out`

Parameters

- `input`: batch tensor in (N, C, H, W) format
- `sigmas`: float tensor with one sigma value per image in the batch
- `max_ksize`: maximum kernel size in pixels
- `out`: output tensor (may be `NULL`)

4.1.6.3 Lighting

The following functions change the lighting of 2D images. Both input and output must be contiguous and in channel-first format (C, H, W) (channel, height, width). All dtypes and an arbitrary number of channels is supported.

Output tensor `out` may be `NULL`, in which case a new tensor of the same shape and dtype as the input is returned. Output tensor must be same shape and dtype as the input. If output is given `NULL` is returned.

CudaTensor *`augpy::lighting`(*CudaTensor* **tensor*, *CudaTensor* **gammagrays*, *CudaTensor* **gammacolors*, *CudaTensor* **contrasts*, double *vmin*, double *vmax*, *CudaTensor* **out*)

Apply lighting augmentation to a batch of images. This is a four-step process:

a. Normalize values $v_{norm} = \frac{v - v_{min}}{v_{max} - v_{min}}$ with v_{max} the maximum lightness value

b. Apply contrast change

c. Apply gamma correction

d. Denormalize values $v' = v_{norm} * (v_{max} - v_{min}) + v_{min}$

To change contrast two reference functions are used. With contrast $\rfloor \geq 0$, i.e., increased contrast, the following function is used:

$$f_{pos}(v) = \frac{1.0037575963899724}{1 + exp(6.279 + v \cdot 12.558)} - 0.0018787981949862$$

With contrast $\rfloor < 0$, i.e., decreased contrast, the following function is used:

$$f_{neg}(v) = 0.1755606108304832 \cdot atanh(v \cdot 1.986608 - 0.993304) + 0.5$$

The final value is $v' = (1 - \rfloor) \cdot v + \rfloor \cdot f(v)$.

Brightness and color changes are done via gamma correction.

$$v' = v^{\gamma_{gray} \cdot \gamma_c}$$

with γ_{gray} the gamma for overall lightness and γ_c the per-channel gamma.

Return new tensor if `out` is `NULL`, else `out`

Parameters

- `tensor`: image tensor in (N, C, H, W) format
- `gammagrays`: tensor of N gamma gray values
- `gammacolors`: tensor of $C \cdot N$ gamma values in the format $\gamma_{1,1}, \gamma_{1,2}, ..., \gamma_{1,C}, \gamma_{2,1}, \gamma_{2,2}, ...$
- `contrasts`: tensor of N contrast values in $[-1, 1]$
- `vmin`: minimum lightness value in images
- `vmax`: maximum lightness value in images
- `out`: output tensor (may be `NULL`)

4.1.6.4 Affine Warp

Functions to apply affine transformations on 2D images.

enum `augpy::WarpScaleMode`

Enum whether to scale relative to the shortest or longest side of the image.

Values:

enumerator WARP_SCALE_SHORTEST

Scaling is relative to the shortest side of the image.

enumerator WARP_SCALE_LONGEST

Scaling is relative to the longest side of the image.

int `augpy::make_affine_matrix`(py::buffer *out*, size_t *source_height*, size_t *source_width*, size_t *target_height*, size_t *target_width*, float *angle*, float *scale*, float *aspect*, float *shifty*, float *shiftx*, float *sheary*, float *shearx*, bool *hmirror*, bool *vmirror*, *WarpScaleMode* *scale_mode*, int *max_supersampling*)

Create a 2×3 matrix for a set of affine transformations. This matrix is compatible with the warpAffine function of OpenCV with the WARP_INVERSE_MAP flag set.

Transforms are applied in the following order:

a. shear

b. scale & aspect ratio

c. horizontal & vertical mirror

d. rotation

e. horizontal & vertical shift

Return recommended supersampling factor for the warp

Parameters

- `out`: output buffer that matrix is written to; must be a writeable 2×3 `float` buffer
- `source_height`: h_s height of the image in pixels
- `source_width`: w_s width of the image in pixels
- `target_height`: h_t height of the output canvas in pixels
- `target_width`: w_t width of the output canvas in pixels
- `angle`: clockwise angle in degrees with image center as rotation axis

- `scale`: scale factor relative to output size; 1 means fill target height or width wise depending on `scale_mode` and whichever is longest/shortest; larger values will crop, smaller values leave empty space in the output canvas
- `aspect`: controls the aspect ratio; 1 means same as input, values greater 1 increase the width and reduce the height
- `shifty`: shift the image in y direction (vertical); 0 centers the image on the output canvas; -1 means shift up as much as possible; 1 means shfit down as much as possible; the maximum distance to shift is $max(scale \cdot h_s - h_t, h_t - scale \cdot h_s)$
- `shiftx`: same as `shifty`, but in x direction (horizontal)
- `sheary`: controls up/down shear; for every pixel in the x direction move `sheary` pixels in y direction
- `shearx`: same as `sheary` but controls left/right shear
- `hmirror`: if `true` flip image horizontally
- `vmirror`: if `true` flip image vertically
- `scale_mode`: if *WARP_SCALE_SHORTEST* scale is relative to shortest side; this fills the output canvas, cropping the image if necessary; if *WARP_SCALE_LONGEST* scale is relative to longest side; this ensures the image is contained inside the output canvas, but leaves empty space
- `max_supersampling`: upper limit for recommended supersampling

void `augpy::warp_affine`(*CudaTensor* *src, *CudaTensor* *dst, py::buffer *matrix*, *CudaTensor* *background*, int *supersampling*)

Takes an image in channels-last format (H, W, C) and affine warps it into a given output tensor in channels-first format (C, H, W). Any blank canvas is filled with a background color. The warp is performed with bi-linear and supersampling.

Parameters

- `src`: image tensor
- `dst`: target tensor
- `matrix`: 2×3 `float` transformation matrix, see *make_affine_matrix* for details
- `background`: background color to fill empty canvas
- `supersampling`: supersampling factor, e.g., 3 means 9 samples are taken in a 3×3 grid

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

P

Q

R

S

T

U

W